Kaucher/Klatte/Ullrich/
Wolff v. Gudenberg

Programmiersprachen im Griff

Band 4: ADA

B·I·
Hochschultaschenbücher
Band 607

FG
44

B.I.-Hochschultaschenbücher
Band 607

Programmiersprachen im Griff

Band 4: ADA

von
Dr. habil. Edgar Kaucher
Dr. Rudi Klatte
Prof. Dr. Christian Ullrich
Dr. Jürgen Frhr. Wolff von Gudenberg
Universität Karlsruhe

Bibliographisches Institut Mannheim/Wien/Zürich
B.I.-Wissenschaftsverlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Kaucher, Edgar:
Programmiersprachen im Griff / von Edgar Kaucher; Rudi Klatte; Christian Ullrich. - Mannheim; Wien; Zürich: Bibliographisches Institut
(BI-Hochschultaschenbücher; . . .)
Bd. 4 u.d.T.: Programmiersprachen im Griff
NE: Klatte, Rudi:; Ullrich, Christian:; HST
Bd. 4. → ADA

ADA / von Edgar Kaucher . . . - Mannheim; Wien; Zürich: Bibliographisches Institut, 1983.
(Programmiersprachen im Griff / von Edgar Kaucher; Rudi Klatte; Christian Ullrich; Bd. 4) (BI-Hochschultaschenbücher; Bd. 607)
ISBN 3-411-00607-2
NE: Kaucher, Edgar [Mitverf.]; 2. GT

Fachhochschule Bochum
-Bibliothek -
86199 A

Alle Rechte, auch die der Übersetzung in fremde Sprachen, vorbehalten. Kein Teil dieses Werkes darf ohne schriftliche Einwilligung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.
© Bibliographisches Institut AG, Mannheim 1983
Druck und Bindearbeit: Hain-Druck GmbH, Meisenheim/Glan
Printed in Germany
ISBN 3-411-00607-2

VORWORT

Mit diesem Band wird unter dem Titel "Programmiersprachen im Griff" eine Reihe von Lehrbüchern über die bekanntesten Programmiersprachen in einheitlicher Gliederung, Gestaltung und Terminologie fortgesetzt.

In diese Reihe gehen Erfahrungen ein, die wir in den letzten Jahren bei der Abhaltung von Vorlesungen über Programmieren für Hörer aller Fachrichtungen an der Universität Karlsruhe, sowie in Seminaren und Arbeitskreisen über höhere Programmiersprachen gewonnen haben. Die Erfahrungen haben gezeigt, daß ein wirkungsvolles Ausbilden und Erlernen nur möglich ist, wenn die Programmiersprachen einheitlich, übersichtlich und einprägsam dargestellt werden. Dies wird erreicht durch die Darstellung der Syntax der Sprachen in einer standardisierten Syntaxdiagrammform und einer darauf abgestimmten Beschreibung der Semantik. Dem Zweck dieser Reihe entsprechend sind die folgenden Punkte besonders berücksichtigt:

- Vollständige Wiedergabe des Inhaltes der jeweiligen originalen Sprachbeschreibungen.
- Einheitliche Darstellung der Syntax, die insbesondere durch den Aufbau der Diagramme, deren Bezeichungsweisen, die Wahl der Kommentare usw. erzielt wird.
- Einheitliche Beschreibungsart der Semantik, insbesondere durch Verwendung gleicher Begriffe, Ausdrucksweisen usw.
- Einarbeitung von Semantikhinweisen in die Syntaxdiagramme zur schnellen Erfassung des Sprachelementes.
- Didaktische Aufbereitung des Stoffes durch den linearen Aufbau von einfachen Sprachelementen bis zum komplizierten Programm.
- Angabe von instruktiven Beispielen.
- Ausführliche Register.

Damit eignet sich jedes Buch dieser Reihe unmittelbar als Vorlesungsmanuskript und dient auch dem Neuling bestens als Leitfaden für das Erlernen der beschriebenen Programmiersprache. Trotz der didaktischen Aufbereitung ist jeweils der volle Sprachumfang wiedergegeben.

Der hier vorliegende Band ist in sich abgeschlossen und kann ohne weitere Voraussetzungen gelesen werden. Die Einarbeitung in die Materie wird jedoch deutlich erleichtert, wenn Kenntnisse in höheren Programmiersprachen, insbesondere von PASCAL, vorliegen. Um den Rahmen nicht zu sprengen, sind alle Beispiele nur exemplarisch zu verstehen und erheben keinen Anspruch auf Vollständigkeit, Optimalität, usw. Insbesondere wird aus Platzgründen auf Flußdiagramme völlig verzichtet.

Unser Dank gilt allen, die zum Gelingen dieses Buches - sei es durch Diskussionen oder durch das Austesten von Programmen - beigetragen haben, insbesondere den Herren cand. math. Martin Hanke, cand. math. Torsten Rössel, cand. math. Günter Schumacher sowie Herrn Dipl. Math. Werner Frangen.

Wir danken Herrn Prof. Dr. Ulrich Kulisch, der das Entstehen dieses Buches ermöglicht hat, und dem Bibliographischen Institut für die Aufnahme dieses Buches in die Reihe "Hochschultaschenbücher".

Besondere Anerkennung verdient Frl. Dipl. Math. Margit Bürzele, die nicht nur mit großer Sorgfalt die mühevolle Niederschrift des Manuskriptes besorgte, sondern auch alle Diagramme in die perfekte Form gebracht hat.

Karlsruhe, Oktober 1982 Die Verfasser

INHALTSVERZEICHNIS

A. EINLEITUNG 12

1. Programmiersprachen - ein kurzer Überblick 12
2. Darstellung von Programmiersprachen 16
   2.1 Alphabet 16
   2.2 Syntax 16
      2.2.1 Elementare Syntaxdiagramme 17
      2.2.2 *T*-Syntaxdiagramme 24
   2.3 Semantik 28
   2.4 Pragmatik 28
3. Hinweise zum Lesen des Buches 30

B. DAS ADA PROGRAMM 31

1. Programmaufbau 31
   1.1 Unterprogramme 37
   1.2 Pakete 39
   1.3 Prozesse 41
   1.4 Untereinheiten 43
2. Vereinbarungen 46
   2.1 Die Vereinbarung von Typen 48
      2.1.1 Die skalaren Typen 50
         2.1.1.1 Die diskreten Typen 50
            2.1.1.1.1 Der Aufzählungstyp 51
            2.1.1.1.2 Der Typ boolean 52
            2.1.1.1.3 Der Typ character 52
            2.1.1.1.4 Der Typ integer 53
         2.1.1.2 Reelle Typen 56
            2.1.1.2.1 Der Gleitpunkttyp 57
            2.1.1.2.2 Der Festpunkttyp 60
      2.1.2 Strukturierte Typen 62
         2.1.2.1 Der Array Typ 62

2.1.2.1.1 Der String Typ 64
2.1.2.2 Der Record Typ 65
2.1.3 Der Zeigertyp 70
2.1.4 Untertypen 73
2.1.5 Abgeleitete Typen 74
2.1.6 Geschützte Typen 76
2.2 Die Vereinbarung von Variablen und Konstanten 78
2.3 Die Vereinbarung von Unterprogrammen 82
2.4 Die Vereinbarung von Paketen 82
2.5 Die Vereinbarung von Unterprogramm- und Paketschemata 83
2.6 Die Vereinbarung von Prozessen 83
2.7 Die Ausnahmevereinbarung 85
2.8 Umbenennungen 85
3. Ausdrücke 88
3.1 Der logische Ausdruck 92
3.2 Der Zeichenausdruck 100
3.3 Der Ausdruck vom Aufzählungstyp 101
3.4 Der ganzzahlige Ausdruck 102
3.5 Der reelle Ausdruck 105
3.5.1 Der Gleitpunktausdruck 105
3.5.2 Der Festpunktausdruck 108
3.6 Der Ausdruck für Felder 110
3.6.1 Der Ausdruck für Felder mit Komponententyp boolean 114
3.6.2 Der Ausdruck für eindimensionale Felder 116
3.6.3 Der Ausdruck für Zeichenketten 117
3.7 Der Record Ausdruck 117
3.8 Der Ausdruck für Zeiger 121
3.9 Das Überladen von Operatoren 123
3.10 Die Typkonvertierung 125
3.11 Attribute 127

4. Anweisungen 128
4.1 Die leere Anweisung 129
4.2 Die Wertzuweisung 130
4.3 Einfache Anweisungen für Ein- und Ausgabe 131
4.4 Die bedingte Anweisung 137
4.5 Die Wiederholungsanweisung und der Sprung aus der Schleife 138
4.6 Marken und die Sprunganweisung 141
4.7 Der Unterprogrammrücksprung 143
4.8 Die Auswahlanweisung (case-Anweisung) 144
4.9 Der Block 146
4.10 Der Aufruf einer Prozedur 147
4.11 Anweisungen zur Synchronisierung von Prozessen 148
4.12 Das Auslösen einer Ausnahme 148
4.13 Die Code Anweisung 148
4.14 Beispiele 149
5. Handhabung von Unterprogrammen 158
5.1 Die Vereinbarung von Unterprogrammen 158
5.2 Der Unterprogrammrücksprung 162
5.3 Der Aufruf einer Prozedur 164
5.4 Der Aufruf einer Funktion 164
5.5 Der Aufruf eines Operators 166
5.6 Das Zusammenspiel der formalen und der aktuellen Argumente 167
5.7 Das Überladen von Unterprogrammen 169
5.8 Beispiele 170
6. Handhabung von Programmschemata 175
6.1 Die Vereinbarung von Schemata 176
6.2 Die Ausprägung von Schemata 181
6.3 Das Zusammenspiel der formalen und der aktuellen Schemaargumente 183
6.4 Beispiele 186

7. Die Behandlung von Ausnahmen 193
7.1 Vordefinierte Ausnahmen 193
7.2 Die Vereinbarung von Ausnahmen 196
7.3 Das Auslösen einer Ausnahme 197
7.4 Der Ausnahmebehandlungsteil 198
7.5 Die Verbreitung von Ausnahmen 200
7.6 Beispiel 206
8. Compilerinstruktionen 208
8.1 Direktiven (Pragmas) 208
8.2 Darstellungsangaben 214
8.2.1 Längenangaben 216
8.2.2 Aufzählungstypdarstellung 218
8.2.3 Recorddarstellung 218
8.2.4 Adreßangaben 219
8.3 Unkontrollierte Speicherfreigabe 220
8.4 Unkontrollierte Typumwandlung 221
9. Handhabung von Prozessen 222
9.1 Die Aktivierung von Prozessen 224
9.2 Das Rendezvous von Prozessen 226
9.2.1 Die Vereinbarung eines Eingangs 227
9.2.2 Der Aufruf eines Eingangs 228
9.2.3 Die Annahmeanweisung (accept-Anweisung) 229
9.2.4 Die Synchronisierung 231
9.3 Das bedingte Rendezvous 232
9.4 Das verzögerte Rendezvous 235
9.5 Die Terminierung von Prozessen 239
9.6 Prioritätsregelung 242
9.7 Gemeinsame Variable 244
9.8 Attribute 245
9.9 Beispiel 247
10. Programmstruktur und Modularität 259
10.1 Die Programmbibliothek 259
10.2 Gültigkeitsbereiche von Vereinbarungen 263

10.3 Sichtbarkeit von Größen eines Programmes 264
10.4 Kontextangaben 268
10.5 Konflikte bei der Namensgebung 271
10.6 Übersetzung und Ausführung eines Programmes 273

11. Ein- bzw. Ausgabe und Handhabung von Files 275
11.1 Das Paketschema input_output 276
11.1.1 Die Verwaltung von Files 276
11.1.2 Ein- bzw. Ausgabe mit Files 281
11.1.3 Ausnahmesituationen 284
11.2 Das Paket text_io 285
11.2.1 Die Verwaltung von Textfiles 286
11.2.2 Ein- bzw. Ausgabe mit Textfiles für 291
11.2.2.1 Zeichen, Zeichenketten 291
11.2.2.2 logische Werte 296
11.2.2.3 ganzzahlige Werte 297
11.2.2.4 Gleitpunktzahlen 299
11.2.2.5 Festpunktzahlen 303
11.2.2.6 Werte von Aufzählungstypen 306
11.3 Das Paket low_level_io 308
11.4 Der Ausdruck für Files 309

C. SYNTAXDIAGRAMME 311

D. VERZEICHNISSE 343
1. Verzeichnis der Syntaxdiagramme 343
2. Verzeichnis der Wortsymbole 347
3. Verzeichnis der Standardnamen 349
4. Das ADA Wörterbuch 353
5. Stichwortverzeichnis 360

E. LITERATUR 377

A. EINLEITUNG

Einführende Bemerkungen zu Sinn und Zweck des Programmierens, zur Problemanalyse und -algorithmisierung, sowie zur Entwicklungsgeschichte, Aufbau und Wirkungsweise elektronischer Rechenanlagen würden den Rahmen dieses Buches sprengen. Sie können ohne weiteres in anderen Bänden der Reihe nachgelesen werden.

1. PROGRAMMIERSPRACHEN - EIN KURZER ÜBERBLICK

Es ist üblich geworden, drei Arten von Programmiersprachen zu unterscheiden:

Maschinensprachen,
maschinenorientierte oder einfache Programmiersprachen (Assemblersprachen),
problemorientierte oder höhere Programmiersprachen.

Maschinensprachen

Jeder Rechenautomat besitzt eine Programmiersprache, die er unmittelbar "lesen" und ohne menschliche Hilfe in Steuersignale umsetzen kann. Sie ist durch seine Bauart festgelegt und i.a. nicht auf Rechner anderen Typs übertragbar. Man bezeichnet sie deshalb auch als Maschinensprache und ein in ihr geschriebenes Programm als M a s c h i n e n p r o g r a m m .

Der Gebrauch von Maschinenprogrammen hat zwei große Nachteile. Einmal muß der Programmierer für jeden Maschinentyp eine

eigene Sprache erlernen, zum anderen benutzen Maschinensprachen fast stets eine Ziffernverschlüsselung. Ein Maschinenprogramm ist deshalb nicht sehr übersichtlich, erfordert aber zu seiner Herstellung ungewöhnlich viel Zeit und Konzentration. Man hat aus diesen Gründen nach Möglichkeiten gesucht, das Programmieren durch Entwicklung einfacherer Sprachen zu erleichtern.

Assemblersprachen (maschinenorientierte Programmiersprachen)

Diese Sprachen benutzen für die einzelnen Rechenoperationen statt einer Ziffernverschlüsselung mnemotechnische Abkürzungen, die sich leicht merken lassen (z.B. mul für Multiplikation); außerdem machen sie Gebrauch von der sog. symbolischen Adressierung, wodurch das Programmieren ganz wesentlich erleichtert wird, und schließlich verwenden sie für den Aufruf häufig vorkommender Unterprogramme sog. Makrobefehle. Sie sind aber sonst den Maschinensprachen noch sehr verwandt, und ihre Anwendung führt nicht zu einer wesentlichen Verkürzung der Programme.

Problemorientierte Programmiersprachen

Gegenüber den Assemblersprachen sind die problemorientierten Programmiersprachen nicht an einen bestimmten Rechnertyp gebunden. Sie verwenden in weitem Maße gebräuchliche mathematische Schreibweisen und lassen sich deshalb besonders leicht erlernen. Zudem verzichten sie fast ganz auf die für das Programmieren so lästigen maschineninternen Operationen, wie z.B. Datentransportvorgänge, die für das eigentliche mathematische Problem gar keine Bedeutung haben. Deshalb nennt man solche Sprachen auch "problemorientiert".

Zur Programmierung mathematischer, naturwissenschaftlicher und technischer Probleme haben u.a. folgende problemorientierte Sprachen weite Verbreitung gefunden:

ALGOL 60 (ALGOrithmic Language)
Die Sprache wurde in Europa gemeinsam von vielen wissenschaftlichen Institutionen aus 6 Ländern entwickelt und 1960 publiziert ([28]).

BASIC (Beginners All-purpose Symbolic Instruction Code)
Ein vereinfachtes, FORTRAN-ähnliches Programmiersystem, besonders auf Kleinrechnern verbreitet. Die Entwicklung ging 1960 von J. Kemmeny und Th. Kurtz aus ([31]).

COBOL (COmmon Business Orientated Language)
Die Sprache ist vorwiegend den Bedürfnissen einer kommerziellen Datenverarbeitung angepaßt und erlaubt in einfacher Weise den Umgang mit großen Datenmengen. Sie wurde 1959 in den USA entwickelt, 1961 erstmals publiziert und seither wiederholt überarbeitet ([39]).

FORTRAN (FORmula TRANslation)
FORTRAN wurde 1954 als erste problemorientierte Programmiersprache von J. W. Backus entworfen, in den USA in mehreren Versionen weiterentwickelt und liegt zur Zeit in dem Standard FORTRAN 77 vor ([29]).

PASCAL
Eine ALGOL-ähnliche, von K. Jensen und N. Wirth Anfang der 70er Jahre entwickelte Programmiersprache mit erweitertem Typ-, Ausdrucks- und Anweisungskonzept. Zwischenzeitlich entstanden auch hier dem modernsten Stand der Wissenschaft angepaßte Versionen, wie z.B. Matrix PASCAL ([17],[18],[30]).

ADA
ADA wurde im Rahmen eines Wettbewerbes des amerikanischen Verteidigungsministeriums entwickelt ([1]). Es soll durch

seine universelle Einsetzbarkeit sowohl im kommerziellen als auch technisch wissenschaftlichen Bereich dazu dienen, die Wartungskosten für Software möglichst gering zu halten. Ein ADA-Programm wird in der Regel auf einem (großen) Rechner entwickelt und getestet und kommt dann auf anderen (kleinen) Rechnern zum Einsatz. ADA ermöglicht einen modularen Aufbau der Programme und getrennte Übersetzung der einzelnen Moduln, dies allerdings bei klarer Definition und Überprüfung der Schnittstellen. Zur Modularisierung stehen Unterprogramme, Pakete und Prozesse zur Verfügung. Pakete fassen zusammenhängende Daten und Operationen zusammen. Prozesse ermöglichen die parallele Verarbeitung.

2. DARSTELLUNG HÖHERER PROGRAMMIERSPRACHEN

Die Festlegung einer höheren Programmiersprache erfolgt gewöhnlich durch vier Komponenten:

Alphabet
Syntax
Semantik
Pragmatik.

2.1 ALPHABET

Grundlage einer jeden Programmiersprache ist ein Symbolvorrat, das sogenannte A l p h a b e t . Mit diesen Symbolen werden Programme dargestellt. Da i.a. nicht jede Symbolfolge als Programm zugelassen werden soll, bedarf es einer Vorschrift, welche die zulässigen Symbolfolgen auswählt. Als Alphabet der deutschen Sprache kann man z.B. alle im Duden aufgelisteten Wörter und Satzzeichen ansehen.

2.2 SYNTAX

Die S y n t a x regelt, welche Symbolfolgen des zugrundegelegten Vokabulars (Alphabets) zulässige "Sätze" (P r o g r a m m e) der Sprache bilden und legt gleichzeitig zu jeder solchen Symbolfolge eine grammatikalische Struktur fest.

Wir machen uns dies an einem Beispiel der deutschen Sprache klar:

Der Satz: "Der Floh beißt den Hund" wird gemäß folgender Tabelle in Sprachelemente zerlegt:

Satzteil	grammatikalische Bezeichnung
Der Floh	Subjekt
beißt	Prädikat
den Hund	Objekt

Entsprechend dieser Zerlegung von natürlichen Sprachen werden auch Programmiersprachen in grammatikalische Bestandteile gegliedert.

Für die Beschreibung der Syntax werden hier Syntaxdiagramme verwendet.

2.2.1 ELEMENTARE SYNTAXDIAGRAMME

Die Gestaltung dieser Diagramme wird in folgender Weise vorgenommen:

- Jedes Diagramm besitzt einen oder mehrere (grammatikalische) B e z e i c h n e r , welche in Großbuchstaben geschrieben und links über das betreffende Diagramm gesetzt sind. Die Bezeichner sind stets so gewählt, daß sie einen Hinweis auf das durch das Diagramm dargestellte Sprachelement enthalten.
- Ein Diagramm ist aufgebaut aus Bezeichnern von Diagrammen, Symbolen des Alphabets und durchgezogenen bzw. gepunkteten Linien.
- Ein Symbol des *Alphabets* ist in einen Kreis eingeschlossen bzw. als Wortsymbol unterstrichen. Folgen von Symbolen sind in ein Oval eingeschlossen.
- Die Verbindung von Bezeichnern und Symbolen des Alphabets geschieht durch durchgezogene bzw. gepunktete Linien.
- Stehen über einem Syntaxdiagramm mehrere Bezeichner, so

bedeutet dies für alle darüberstehenden Bezeichner den gleichen syntaktischen Aufbau.

- Die Diagramme sind durchnumeriert.
  z.B. bedeutet

  ADA 65: NAME
  MARKE

  daß die Bezeichner NAME und MARKE das gemeinsame Diagramm mit der Nummer 65 haben, d.h. syntaktisch gleich aufgebaut sind.
- Die Diagramme sind mit in Kursivschrift geschriebenen erläuternden Kommentaren versehen. Sie erleichtern das semantische Erfassen des betreffenden Sprachelementes und sind entweder unter, neben oder auch über der entsprechenden Diagrammzeile angebracht.
- Am Schluß wird eine Liste aller vorkommenden Bezeichner in alphabetischer Anordnung gegeben, die das Auffinden der zugehörigen Diagramme erleichtert (siehe D.1).

Die Verwendung der Syntaxdiagramme zum Schreiben von Programmen geschieht in der folgenden Weise:

- Der *Durchlauf* eines Diagrammes beginnt am linken oberen Eingang.
- Durchgezogene Linien werden von links nach rechts bzw. von oben nach unten, gepunktete Linien in umgekehrter Richtung *durchlaufen*.
- Der *Durchlauf* eines Diagrammes ist beendet, wenn das Diagramm am rechten unteren Ende verlassen wird.
- Treten beim *Durchlauf* eines Diagrammes Symbole des Alphabets A auf, so werden diese zum Zeitpunkt des Auftretens notiert und der *Durchlauf* fortgesetzt.
- Tritt beim *Durchlauf* eines Diagrammes ϕ eine Syntaxvariable ψ auf (wobei ψ gleich ϕ zulässig), so ist an dieser Stelle der *Durchlauf* von ϕ zu unterbrechen, das

mit diesem Namen ψ bezeichnete Diagramm zu *durchlaufen* und danach der *Durchlauf* des Diagrammes ϕ an der Stelle fortzusetzen, an der er unterbrochen wurde.

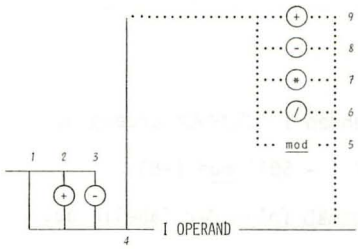
- Zur Produktion eines Programmes ist stets das mit dem Startbezeichner PROGRAMM versehene Diagramm zu *durchlaufen*.

Grundsätzlich kann mit der oben beschriebenen Vorgehensweise aus dem Startbezeichner PROGRAMM ein gewünschtes Programm geschrieben werden. Dieses Vorgehen ist jedoch schwerfällig und wird man nur zur Einarbeitung verwenden. Tatsächlich prägt sich der Leser durch wiederholten Gebrauch der Syntaxdiagramme schnell den Aufbau der einzelnen Sprachelemente (Syntaxdiagramme) ein und kann somit einen in einem Diagramm ϕ auftretenden Syntaxbezeichner ψ unmittelbar durch die gewünschte Symbolfolge bzw. einen zulässigen Diagrammteil ersetzen, ohne zuvor ψ (und deren Folgediagramme) durchlaufen zu müssen.

Beispiel:

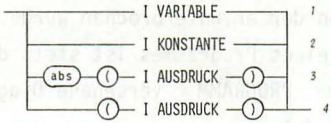
Mit den Diagrammen aus

ADA 28: I AUSDRUCK



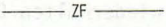
aus

ADA 31: I OPERAND



aus

ADA 39: I KONSTANTE (I K)

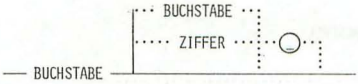


aus

ADA 33: I VARIABLE (I VAR)

aus

ADA 65: I VAR NAME



können wir z.B. folgenden I AUSDRUCK erzeugen:

((i + j) * k / l - 5Ø1) <u>mod</u> (-8) .

Die Erzeugung läuft gemäß folgender Tabelle ab.

D Nr.	Knoten Nr.	Sprachelement
28	1,4,5,4	I OPD <u>mod</u> I OPD
31	4	(I A) <u>mod</u> I OPD
28	1,4,7,4 6,4,8,4	(I OPD * I OPD / I OPD - I OPD) <u>mod</u> I OPD
31	4	((I A) * I OPD ...
28	1,4,9,4	((I OPD + I OPD) * ...
31	1	((I VAR + I OPD) * ...
31	1	((I VAR + I VAR) * I OPD / I OPD - I OPD) <u>mod</u> I OPD
31	1	((...) * I VAR / I OPD - I OPD) <u>mod</u> I OPD
31	1	((...) * I VAR / I VAR - I OPD) <u>mod</u> I OPD
31	2	((... - I K) <u>mod</u> I OPD
31	4	((... <u>mod</u> (I A)
28	3,4	((... <u>mod</u> (-I OPD)
31	2	((... <u>mod</u> (- I K)

Unter Verwendung der Diagramme 33, 65 und 68 kann schließlich I VAR ersetzt werden durch i,j,k,l; mit den Diagrammen 39,56 und 67 kann I K ersetzt werden durch 5Ø1 bzw. 8.

Umgekehrt tritt oft das Problem auf, zu prüfen, ob ein vorgegebenes Sprachelement grammatikalisch korrekt ist. Dabei ist zu versuchen, den obigen Prozeß in umgekehrter Richtung zu durchlaufen. Gelingt es, den zu prüfenden Ausdruck auf einen I AUSDRUCK zurückzuführen, so ist das Sprachelement korrekt, andernfalls ist es grammatikalisch unzulässig.

Die Durchführung dieses Prozesses (Syntaxanalyse) erfordert

Geschick und Vertrautheit mit den Sprachelementen. Daher ist es für den Programmieranfänger einfacher, das Sprachelement in der zuerst beschriebenen Weise zu erzeugen. Gelingt dies, so ist das vorgelegte Sprachelement grammatikalisch korrekt, andernfalls grammatikalisch unzulässig.

Beispiel: Ist i * - n ein korrekter I AUSDRUCK ?

	Diagramm Nr.	Knoten Nr.	Sprachelement
	28	1,4,7,4	I OPD * I OPD
oder	28	1,4,8,4	I OPD - I OPD

In beiden Fällen kann der zweite Operator nicht unmittelbar durch Ersetzen von I OPD erzeugt werden. Dies erkennt man durch folgende Versuche: da in I OPD explizit weder das Zeichen - noch * vorkommt, ist dies höchstens über 4 mittels I AUSDRUCK möglich. In diesem Fall müßte dann allerdings geklammert werden, was nicht auf den vorgegebenen Ausdruck führt:

Fährt man im ersten Fall mit ADA 31,4 und ADA 28,3,4 fort, so ergibt sich

i * (-n)

bzw. im zweiten Fall mit ADA 31,4 und ADA 28,1,4,7,4

(i * I OPD) - n ,

d.h. das Zeichen * ist nicht allein erzeugbar.

Im Gegensatz dazu ist jedoch

-n * i

grammatikalisch korrekt, wie der Leser selbst nachvollziehen kann.

Die arbeitsaufwendige Syntaxanalyse übernimmt die Rechenanlage, die vor Ausführung eines eingegebenen Programmes automatisch die Analyse durchführt und fehlerhafte grammatikalische Konstruktionen meldet.

Abschließend erläutern wir die Erzeugung und das Erlernen des syntaktischen Aufbaues von Sprachelementen an folgendem

Beispiel:

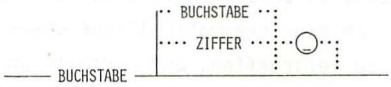
Mit den Syntaxdiagrammen

ADA 65: NAME

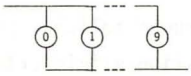
AUSNAHME NAME
BIBLIOTHEKS EINHEITS NAME (BIBL EINH NAME)
BLOCKNAME (BL NAME)
τ DISKRIMINANTEN NAME (τ DISKR NAME) $\tau \in \Theta_D$
ENTRY FAMILIEN NAME (ENTRY FAM NAME)
ENTRY NAME
τ FUNKTIONS NAME (τ FKT NAME) $\tau \in \Theta \cup \Phi$
GERAET
τ KOMPONENTEN NAME (τ KOMP NAME) $\tau \in \Theta \cup \{PZS\} \cup \Phi$
τ KONSTANTEN NAME (τ KONST NAME) $\tau \in \Theta$
MARKE
PAKET NAME (PAK NAME)
PROZEDUR NAME (PZD NAME)
SCHLEIFEN NAME (SCHL NAME)
τ TYP NAME $\tau \in \Theta \cup \{PZS\} \cup \Phi$
τ VARIABLEN NAME (τ VAR NAME) $\tau \in \Theta \cup \{PZS\} \cup \Phi$

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$, $\Phi = \{IF,OF,IOF,ITF,OTF\}$

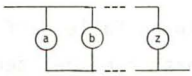
$\Theta_D = \{B,CH,AZ,I\}$



ADA 67: ZIFFER



ADA 68: BUCHSTABE



wird das Schreiben von Namen geregelt. Demnach beginnt ein Name stets mit einem Buchstaben, dem in beliebiger Reihen-

folge Buchstaben oder Ziffern folgen können. Dabei darf zwischen zwei Buchstaben oder Ziffern eines Namens ein Unterstrich _ stehen.

grammatikalisch

zulässig	unzulässig
b1	1Ø5
epsilon	vorsicht!
zins1	1zins
zins_fuss	_zinsfuss_

2.2.2 *T*-SYNTAXDIAGRAMME

Wie wir gesehen haben, wird eine exakte und übersichtliche Beschreibung einer Programmiersprache durch die Notation der Syntax in Syntaxdiagrammen erreicht. Um eine möglichst kompakte Schreibweise zu erzielen, werden diese jedoch nicht nur in der bisher beschriebenen einfachsten Form, sondern auch in k o l l e k t i v e r , s e l e k t i v e r und m e h r s c h i c h t i g e r Schreibweise als sogenannte *T*-Diagramme verwendet.

T-Diagramme erlauben es, durch die Verwendung von *T*-Syntaxvariablen, einerseits ohne Rücksicht auf die Typkennzeichnung (für *T*), sich eine grammatikalische Übersicht über Sprachelemente zu verschaffen, andererseits aber auch eine grammatikalisch korrekte Produktion von Sprachelementen typgebunden (mit Berücksichtigung der jeweils für *T* gegebenen Vorschrift) durchzuführen.

Im letzteren Falle darf also grundsätzlich eine *T*-Syntaxvariable erst dann bei der Produktion eliminiert werden, wenn für *T* ein entsprechend zugelassener Typ festgelegt und eingesetzt wurde bzw. als zulässig befunden wurde.

Es treten daher zwei mögliche Fälle auf: Tritt eine *T*-Syntaxvariable als Diagrammbezeichner (Überschrift) auf, so handelt es sich um ein mehrschichtiges Diagramm (a). Tritt jedoch eine *T*-Syntaxvariable innerhalb eines Diagrammes auf, so handelt es sich entweder um einen kollektiven (b) oder selektiven (c) Diagrammzweig. In diesem Falle (b) oder (c) darf also bei einer Produktion erst dann eliminiert werden, nachdem für *T* ein Typ eingesetzt wurde bzw. *T* den vorgeschriebenen Typ darstellt.

(a) Mehrschichtige Diagramme:

Diagramme, die sich nur nach dem Typ unterscheiden (also nach statischer Semantik), werden zu einem mehrschichtigen Diagramm mit einer *T*-Syntaxvariable zusammengefaßt (Metasprache für Syntaxvariable, zweischichtige Grammatik). In dem Diagramm

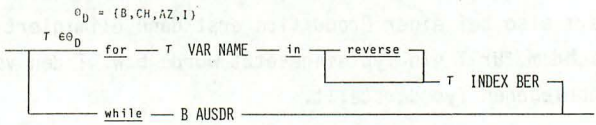
```
ADA 28:  T AUSDRUCK  (T AUSDR, T A)    T∈Θ∪Φ
         Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} ,  Φ = {IF,OF,IOF,ITF,OTF}

————————— T REL ————————————————————————
```

werden durch die Vorschrift $T \in \Theta \cup \Phi$ 15 elementar notierte Diagramme festgelegt. Zum Beispiel erhält man für $T = G$ das Diagramm:

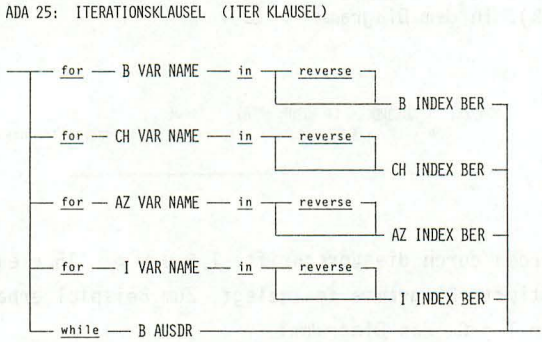
```
ADA 28:  G AUSDRUCK

———— G REL ————————————
```

(b) Kollektive Diagramme:

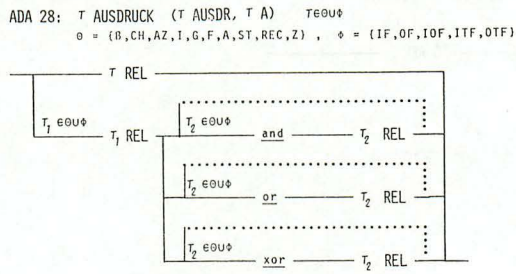
Die Verwendung von T-Syntaxvariablen tragen zur wesentlichen Abkürzung der Diagrammschreibweise bei. In dem Diagramm



werden durch die Vorschrift $T \in \Theta_D$ (Θ_D heißt "Kollektor") 4 gleichstrukturierte Diagrammzeilen in einer einzigen Zeile zusammengefaßt. In ausführlicher Schreibweise erhält das Diagramm die Form:



In einem Diagramm können an verschiedenen Stellen Kollektoren auftreten:



Wir finden hier die beiden Kollektorangaben $T_1 \in \Theta \cup \Phi$ und $T_2 \in \Theta \cup \Phi$ vor. Beim Durchlauf an diesen Kollektorangaben ist jeweils ein Typ T_1 bzw. T_2 aus dem zugehörigen Kollektor $\Theta \cup \Phi$ zu wählen. Steht also eine Kollektorangabe an einem Diagrammzweig innerhalb einer Rekursion, so ist für diesen Diagrammzweig bei jedem Durchlauf ein Typ aus dem Kollektor zu wählen (natürlich unabhängig von vorangegangenen und nachfolgenden Festlegungen). Steht dagegen die Kollektorangabe vor der Rekursion, so ist natürlich die erstmalige Wahl für alle Durchläufe der Rekursion bindend.

(c) Selektive *T*-Diagramme:

Diese Schreibweise erlaubt das Zusammenfassen von semantisch gleichartigen, jedoch syntaktisch unterschiedlichen Sprachelementen zu einem *T*-Diagramm. Wir verwenden in den Diagrammzeilen sogenannte S e l e k t o r e n der Form

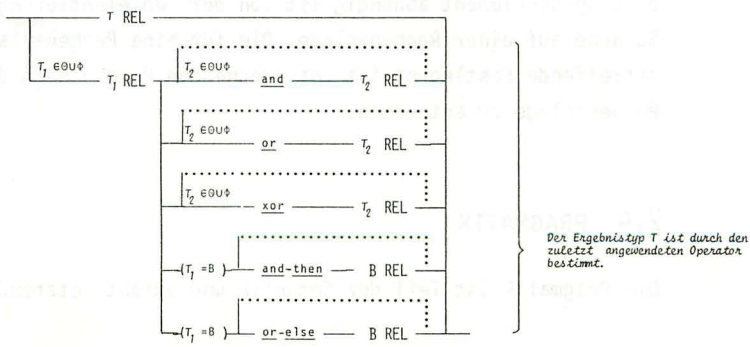
—($T \in$ {B,CH,AZ})—

die den Durchlauf an dieser Stelle nur dann freigeben, wenn eine Produktion für einen aufgelisteten Typ durchgeführt wird.

In dem Diagramm

ADA 28: *T* AUSDRUCK (*T* AUSDR, *T* A) $T \in \Theta \cup \Phi$

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}



besitzen die beiden letzten Zeilen Selektoren mit T_1 =B. Diese Zeilen können also nur durchlaufen werden, wenn zuvor für T_1 REL der Typ B aus dem Kollektor $\Theta \cup \Phi$ gewählt wurde.

2.3 SEMANTIK

Die Semantik gibt zu jedem Programm der Sprache eine Bedeutung an. Im Sinne der Datenverarbeitung sagt die Bedeutung darüber etwas aus, wie die eingegebenen Daten verarbeitet bzw. Daten erzeugt werden. Der Einfachheit halber beschreibt man die Bedeutung von Sprachelementen entsprechend ihrem grammatikalischen Aufbau. Die Beschreibung eines Programmes setzt sich im wesentlichen aus den Bedeutungen der verwendeten Sprachelemente zusammen.

Die Semantik läßt sich nicht in Diagrammform beschreiben, sondern kann hier sinnvoll nur umgangssprachlich angegeben werden. Bei der Beschreibung ist zu beachten, daß die Bedeutung der Sprachelemente sehr streng formuliert ist, um Mehrdeutigkeiten zu vermeiden. Es kommt daher oft auf jedes Wort und auf die Wortstellung an.

Der öfter auftretende Begriff "maschinenabhängig" besagt, daß der damit beschriebene Vorgang bzw. das damit beschriebene Sprachelement abhängig ist von der Implementierung der Sprache auf einer Rechenanlage. Die für eine Rechenanlage zutreffende Festlegung ist entsprechenden Handbüchern der Rechenanlage zu entnehmen.

2.4 PRAGMATIK

Die Pragmatik ist Teil der Semantik und umfaßt letztendlich

den Teil der Sprache, der als allgemein bekannt vorausgesetzt und daher nicht weiter erläutert wird. Beispielsweise wird bei oben genanntem I AUSDRUCK die Bedeutung der Zeichen +, -, (,) usw. nicht näher erläutert, sondern als jedem geläufig vorausgesetzt.

3. HINWEISE ZUM LESEN DES BUCHES

Zur Schreibweise der einzelnen Abschnitte ist folgendes zu bemerken:

- Innerhalb eines Abschnittes bezeichnen in G r o ß - b u c h s t a b e n geschriebene Worte Syntaxvariable, die Bezug nehmen auf Teile des gerade behandelten Diagrammes. Wird dagegen das zugehörige Diagramm selbst angesprochen, so ist stets die entsprechende Diagrammnummer nachfolgend angegeben.
- K u r s i v g e s c h r i e b e n e Wortfolgen bezeichnen die in der gleichen Weise gekennzeichneten Teile in den Syntaxdiagrammen. (Dies trifft jedoch nicht für die erläuternden Kommentare in den Beispielen zur Semantik zu).

  Zur Hervorhebung werden solche Wortfolgen auch zusätzlich gesperrt geschrieben.
- G e s p e r r t u n d n i c h t k u r s i v geschriebene Wortfolgen geben die Stelle der Definition dieses Begriffes an.
- Zitate von Abschnitten ohne Kapitelangabe beziehen sich auf das Kapitel, in dem das Zitat steht.
- Literaturverweise werden in der üblichen Form durch eine Nummer in eckigen Klammern gegeben (z.B. [5]). Der entsprechende Titel ist im Literaturverzeichnis E. zu finden.
- Viele Beispiele in den laufenden Abschnitten sind vollständigen und umfassenden Programmen entnommen, welche gesammelt am Ende eines Kapitels aufgeführt sind.

B. DAS ADA PROGRAMM

1. PROGRAMMAUFBAU

Ein *Programm* besteht aus einer oder mehreren parallelen oder geschachtelten Programmeinheiten. Es gibt 3 verschiedene Arten solcher Einheiten, nämlich Unterprogramme, Pakete und Prozesse. Im einfachsten Fall besteht ein Programm nur aus einem Unterprogramm, einer sogenannten Prozedur.

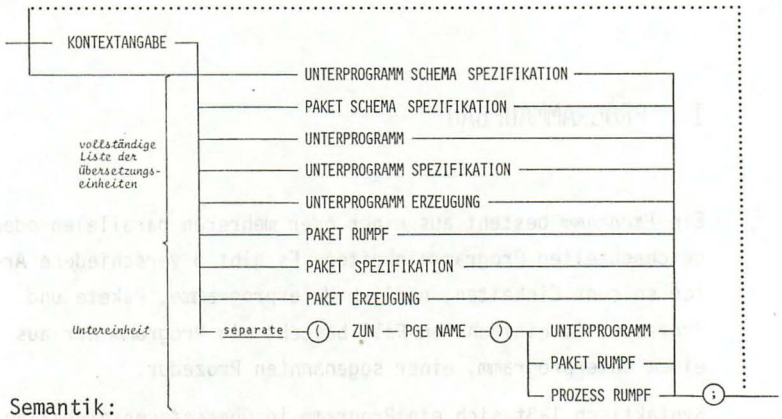
Syntaktisch läßt sich ein Programm in *Übersetzungseinheiten* unterteilen. Ein UNTERPROGRAMM, eine UNTERPROGRAMMSPEZIFIKATION, ein PAKETRUMPF oder eine PAKETSPEZIFIKATION sind Übersetzungseinheiten.

Ferner können auch Unterprogramm- und Paketschemata vereinbart werden, aus denen durch Angabe von Argumenten aktuelle Unterprogramme oder Pakete erzeugt werden. Auch diese Schemaspezifikationen oder Erzeugungen sind Übersetzungseinheiten.

Der Rumpf eines Unterprogrammes, Paketes oder Prozesses läßt sich als *Untereinheit* formulieren und bildet dann auch eine Übersetzungseinheit.

Syntax:

ADA 1: PROGRAMM



Semantik:

Jede Übersetzungseinheit besitzt einen Namen. Die Namen aller Übersetzungseinheiten eines Programms sind verschieden zu wählen mit der Ausnahme, daß ein Unterprogramm den gleichen Namen wie die Unterprogrammspezifikation und ein Paketrumpf den gleichen Namen wie die Paketspezifikation tragen muß.

Im allgemeinen wird ein Programm nicht als ganzes übersetzt werden, sondern die einzelnen Übersetzungseinheiten jeweils für sich. Dabei kann mittels einer KONTEXTANGABE auf andere Übersetzungseinheiten, die in einer Bibliothek vorhanden sind, Bezug genommen werden. Die beiden Übersetzungseinheiten UNTERPROGRAMM und UNTERPROGRAMMSPEZIFIKATION sowie PAKETRUMPF und PAKETSPEZIFIKATION werden durch ihren gemeinsamen Namen gleichzeitig sichtbar. Untereinheiten sind keine Bibliothekseinheiten (siehe 10).

Es darf kein Unterschied bestehen, ob die Übersetzungseinheiten eines Programms getrennt oder gemeinsam übersetzt werden. Wird ein Programm gemeinsam übersetzt, so gilt die Reihenfolge der auftretenden Übersetzungseinheiten. Alle in Kontext-

angaben vorkommenden Übersetzungseinheiten müssen zu diesem Zeitpunkt bereits verarbeitet sein. Man beachte, daß ein Rumpf einer Programmeinheit nach der entsprechenden Spezifikation übersetzt wird, und Untereinheiten nach der entsprechenden Einheit.

Bei getrennter Übersetzung werden die bereits übersetzten Übersetzungseinheiten in der Reihenfolge wieder herangezogen, in der sie in Kontextangaben der zu übersetzenden oder bereits herangezogenen Übersetzungseinheiten auftreten.

Es muß sich also eine Ordnung herstellen lassen, in der alle Übersetzungseinheiten eines Programms gemeinsam übersetzt werden könnten (vgl. 10).

Die A u s f ü h r u n g eines Programmes beginnt mit dem systemabhängigen Anstoß der Ausführung eines Unterprogrammes. Dieses Unterprogramm übernimmt also für diese Ausführung die Rolle eines Hauptprogrammes im üblichen Sinne.

Die Ausführung einer Programmeinheit beginnt mit der Verarbeitung der Vereinbarungen und Definitionen, sodann wird der Anweisungsteil ausgeführt, beginnend mit der ersten Anweisung. Nach Ausführung einer Anweisung wird deren Nachfolgeanweisung ausgeführt. Dabei ist die zu einer Anweisung α gehörige N a c h f o l g e a n w e i s u n g β entweder die durch die Anweisung α ausdrücklich als nachfolgend festgelegte Anweisung β oder andernfalls die physikalisch nachfolgende Anweisung. Die Ausführung einer Programmeinheit wird durch Ausführung einer <u>return</u>-Anweisung oder durch Erreichen des Endes des Anweisungsteils beendet.

<u>Regeln zum Schreiben von Programmen:</u>

(1) Ein Programm wird zeilenweise geschrieben. Am Ende einer Zeile kann, durch -- eingeleitet, ein Kommentar stehen.

KOMMENTAR

—(--)— TEXT — Zeilenende —

TEXT ist dabei eine beliebige Zeichenfolge.

Der verwendete Zeichensatz sollte alle Großbuchstaben und Ziffern, das Leerzeichen und folgende Sonderzeichen (bzw. die angegebenen Ersatzzeichen) enthalten:

+ - * / & ( ) . : ; = < >
' _ | bzw. ! " bzw. % # bzw. :

(2) Innerhalb von Wortsymbolen, Namen und Zahlen dürfen keine Trennzeichen stehen. Dabei versteht man unter einem Trennzeichen ein Leerzeichen oder ein Zeilenendezeichen.

(3) Zwischen zwei unmittelbar aufeinanderfolgenden Namen, Zahlen oder Wortsymbolen muß mindestens ein Trennzeichen stehen.

(4) Bis auf die Regeln (2) und (3) können Trennzeichen beliebig zur übersichtlichen Gestaltung des Programmes verwendet werden. Trennzeichen haben außer ihrer Trennfunktion keine Bedeutung für die Ausführung des Programmes.

(5) Bei der Identifizierung von Namen und Wortsymbolen wird kein Unterschied zwischen entsprechenden Groß- und Kleinbuchstaben gemacht.

(6) Für einen Namen sind alle verwendeten Zeichen signifikant.

(7) Wortsymbole dürfen nicht als Namen verwendet werden. Sie werden wie Namen geschrieben. In diesem Buch sind sie zur Unterscheidung unterstrichen.

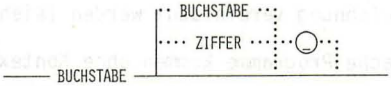
(8) Reservierte Namen, wie z.B. integer dürfen in einer Programmeinheit neu definiert werden.

Ein Verzeichnis der Wortsymbole und reservierten Namen findet sich in D.2 und D.3.

Ein Name hat die folgende Form.

Syntax:

ADA 65: NAME



Ein Name dient zur Identifizierung der durch ihn bezeichneten Größe. Man beachte die Regeln (5) bis (8) zum Schreiben von Programmen.

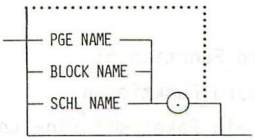
Hinweis:

In den Syntaxdiagrammen tritt NAME nie allein auf, sondern stets mit einem Hinweis auf den Verwendungszweck, z.B. I TYPNAME oder UPG NAME (Unterprogrammname).

Sollen Namen aus anderen Programmeinheiten oder Blöcken verwendet werden, so werden sie mit dem Namen dieser Einheit als Präfix versehen. Solche Einheitsnamen heißen hier ZUNAMEN.

Syntax:

ADA 58: ZUNAME (ZUN)



Semantik:

Ein Zuname ist stets einem Namen vorangestellt. Er bezeichnet die Einheit, in der der Name sichtbar ist. Genaueres folgt in 10.3 .

Durch eine KONTEXTANGABE können einer Übersetzungseinheit Größen aus anderen Bibliothekseinheiten zugänglich gemacht und deren Bezeichnung vereinfacht werden (siehe 10).

Nur sehr einfache Programme kommen ohne Kontextangaben aus. Bereits für Ein- und Ausgabe muß auf entsprechende Standardpakete durch eine Kontextangabe hingewiesen werden (siehe 4.3 oder 11).

Beispiel 1: Newtonverfahren

Bestimmung einer Nullstelle der Funktion $f(x) = \sin\frac{1}{2}x + \frac{x^3-1}{x^2+1}$

mit der Ableitung $f'(x) = \frac{1}{2}\cos\frac{1}{2}x + \frac{x^4+3x^2+2x}{(x^2+1)^2}$ nach dem Newtonverfahren: x_0 beliebiger Startwert

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad , k = 0,1,\ldots .$$

Es soll solange iteriert werden, bis $\left|\frac{x_{k+1}-x_k}{x_k}\right| < \varepsilon$ ist.

Dabei sind x_0 und ε einzulesen, x_{k+1} ist für jeden Schritt auszudrucken.

Version 1:

```
-- Eine Prozedur, die 2 Funktionen enthält
-- außerdem 2 Pakete, die als Bibliothekseinheiten
-- vorhanden seien
with Standard_EA, Standard_Funktionen;
   use Standard_EA, Standard_Funktionen;
      -- Standard_EA ist ein Paket mit Ein- und Ausgabeproze-
      -- duren für die Standardtypen (s. 4.3)
```

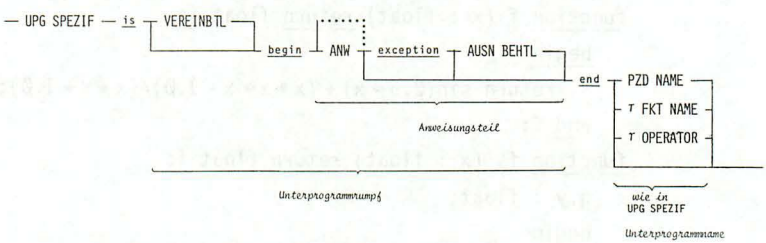
```
    -- Standard_Funktionen ist ein Paket für die Berechnung
    -- arithmetischer Standardfunktionen (s. 1.2)
  procedure Newton is
    xk,xk1,eps  :  float;
    function f (x : float) return float is
      begin
        return sin(0.5 * x) + (x * x * x - 1.0)/(x * x + 1.0);
      end f;
    function fs (x : float) return float is
      q,y : float;
      begin
        q:= x * x; y:= q + 1.0;
        return 0.5 * cos(0.5 * x) - (q * q + 3.0 * q + 2.0 * x)/
                                                    (y * y);
      end fs;
  begin        -- Newton
    get(xk1);
    get(eps);
Iteration: loop
            xk:= xk1;
            xk1:= xk - f(xk)/fs(xk);
            put(xk1);
            exit when abs((xk1 - xk)/xk) < eps;
          end loop Iteration;
  end Newton;
```

1.1 UNTERPROGRAMME

Als Unterprogramme stehen wie üblich Prozeduren und Funktionen zur Verfügung. Als spezielle Funktionen lassen sich zusätzlich Operatoren vereinbaren.

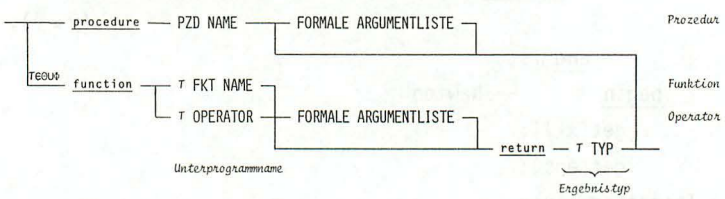
Syntax:

ADA 6: UNTERPROGRAMM (UPG)



ADA 7: UNTERPROGRAMM SPEZIFIKATION (UPG SPEZIF)

O = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}



Semantik:

Bei einer Vielzahl von Unterprogrammen läßt sich die Lesbarkeit des Programmes erhöhen durch eine zusätzliche, eigenständige UNTERPROGRAMMSPEZIFIKATION. Sie muß mit der im Unterprogramm stehenden Spezifikation identisch sein bis auf mögliche Unterschiede in der Schreibweise der Namen der formalen Argumente. Falls sich zwei Unterprogramme gegenseitig aufrufen, ist eine eigenständige Unterprogrammspezifikation unumgänglich.

Die Verarbeitung einer Unterprogrammspezifikation wird, auch wenn diese wiederholt wird, nur einmal vorgenommen. Sie beginnt mit der Einführung des Namens, dann werden die formalen Argumente in der Reihenfolge ihres Auftretens verarbeitet, bei Funktionen folgt die Verarbeitung des Resultattyps.

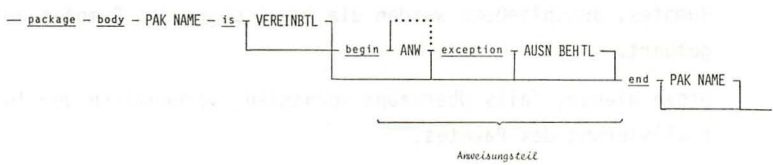
Die Ausführung eines Unterprogrammes wird durch den Aufruf bewirkt. Die Ausführung beginnt nach der Argumentübergabe, wie bereits erwähnt, mit der Verarbeitung des Vereinbarungsteils. Anschließend werden die Anweisungen ausgeführt.

1.2 PAKETE

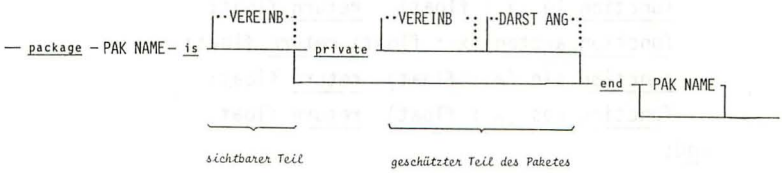
Ein Paket besteht in der Regel aus der Paketspezifikation und dem Paketrumpf.

Syntax:

ADA 9: PAKETRUMPF



ADA 10: PAKETSPEZIFIKATION (PAK SPEZIF)



Semantik:

Ein Paket dient zur Sammlung von Konstanten, Datentypen und Unterprogrammen. Falls keine Unterprogramme auftreten, kann ein Paket nur aus der Paketspezifikation bestehen. Im sichtbaren Teil stehen die Vereinbarungen und Spezifikationen, die an anderen Stellen des Programmes verwendet werden dürfen.

Im geschützten Teil steht die ausführliche Beschreibung der geschützten Typen. Im Rumpf sind die Unterprogramme vereinbart, deren Spezifikationen im sichtbaren Teil aufgeführt sind. Außerdem können hier wie auch im geschützten Teil weitere Größen (Typen, Variable, Unterprogramme,...) vereinbart werden, auf die an anderen Stellen des Programmes nicht zugegriffen werden kann.

Die Verarbeitung eines Paketes beginnt mit der Verarbeitung der Paketspezifikation, es folgt der Vereinbarungsteil des Rumpfes, anschließend werden die Anweisungen des Rumpfes ausgeführt.

Diese dienen, falls überhaupt vorhanden, vornehmlich der Initialisierung des Paketes.

Beispiel: Paket Standardfunktionen

```
package Standard_Funktionen is
   function sqrt (x : float) return float;
   function exp (x : float)  return float;
   function ln (x : float)   return float;
   function arctan (x : float) return float;
   function sin (x : float)  return float;
   function cos (x : float)  return float;
end;

package body Standard_Funktionen is
   type long is digits 20;
   function sincos (x : float; b : boolean) return float is
      function argument_reduction (x : float) return long;
```

```
      function approx (y : long) return long is
         begin    -- Berechnung einer Approximation für sin
            ...   -- bzw. cos
         end;
   function sin (x : float) return float is
      begin
         return sincos (x,true);
      end;
   function cos (x : float) return float is
      begin
         return sincos (x,false);
      end;
   ...
end;
```

1.3 PROZESSE

Während alle anderen Programmeinheiten sequentiell ablaufen, bieten Prozesse die Möglichkeiten zur Parallelarbeit. Alle in einer Übersetzungseinheit vereinbarten Prozesse werden zu Beginn des Anweisungsteils dieser Übersetzungseinheit gleichzeitig aktiviert und führen ihre Anweisungsfolgen parallel, d.h. unabhängig voneinander, aus. Zur Synchronisierung von Prozessen dienen spezielle Anweisungen (siehe 9).

Beispiel 1: Version 2

Die Berechnung von f(x) und f'(x) geschieht durch zwei parallele Prozesse.

```
with Standard_EA, Standard_Funktionen;
   use Standard_EA, Standard_Funktionen;
   procedure Newton is
      fsxk,fxk,xk,xk1,eps : float;
      task f is
```

```
    entry arg (x : in float);
    entry wert (x : out float);
end;
task body f is
    z : float;
    begin loop
            accept arg (x : in float) do
                z:= x;
            end;
            z:= sin(Ø.5 * z) + (z * z * z - 1.Ø)/(z * z +1.Ø);
            select accept wert (x : out float) do
                    x:= z;
                end;
            or terminate;
            end select;
        end loop;
    end f;
task fs is
    entry arg (x : in float);
    entry wert (x : out float);
end;
task body fs is
    y,q,z : float;
    begin loop
            accept arg (x : in float) do
                z:= x;
            end;
            q:= z * z;
            y:= q + 1.Ø;
            z:= Ø.5 * cos(Ø.5 * z) - (q * q + 3.Ø * q +
                                    2.Ø * z)/(y * y);
            select accept wert (x : out float) do
                    x:= z;
                end;
```

```
                or terminate;
                end select;
            end loop;
        end fs;
    begin     -- Newton
        get(xk1);
        get(eps);
Iteration: loop
            xk:= xk1
            f.arg(xk);
            fs.arg(xk);      -- Start der parallelen
                             -- Berechnung von f und fs
            f.wert(fxk);
            fs.wert(fsxk);
            xk1:= xk - fxk/fsxk;
            put (xk1);
            exit when (abs(xk1 - xk)/xk) < eps;
        end loop Iteration;
    end Newton;
```

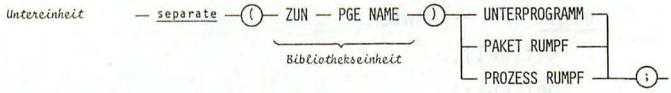
-- Durch andere Aufteilung der Schleife
-- (Aufruf von f. und fs.arg unmittelbar nach Aufruf von
-- wert und einmal vor der Schleife) erhöht sich die Ge-
-- schwindigkeit, da während der Funktionsberechnung die
-- Iterationsberechnung durchgeführt wird.

1.4 UNTEREINHEITEN

Der Rumpf eines Unterprogrammes, Paketes oder Prozesses kann als Untereinheit getrennt übersetzt werden. Dadurch erhöht sich die Übersichtlichkeit, da die Vereinbarungen dichter zusammenstehen, außerdem lassen sich so gegenseitige Aufrufe formulieren.

Syntax:

Aus ADA 1: PROGRAMM



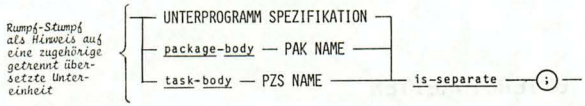
Untereinheiten lassen sich schachteln. Die zu einer Untereinheit unmittelbar übergeordnete Einheit heißt M u t t e r - e i n h e i t .

Die Namen aller (auch der geschachtelten) Untereinheiten einer Bibliothekseinheit müssen verschieden sein, während verschiedene Bibliothekseinheiten Untereinheiten gleichen Namens besitzen dürfen.

Im Vereinbarungsteil der Muttereinheit muß als Hinweis auf die Untereinheit der Rumpf in rudimentärer Form (als Rumpfstumpf) vereinbart werden.

Syntax:

Aus ADA 17: VEREINBARUNGSTEIL (VEREINBTL)



Der Untereinheit sind alle Bibliothekseinheiten, die der Muttereinheit sichtbar sind, ebenfalls sichtbar, so als ob sie anstelle des Rumpfstumpfes stünde. Zusätzlich können mittels Kontextangaben weitere sichtbar gemacht werden.

Beispiel 1: Version 3

Newtonverfahren mit Untereinheiten.

```
with Standard_EA, Standard_Funktionen;
   use Standard_EA, Standard_Funktionen;
   procedure Newton is
      xk,xk1,eps : float;
      function f (x : float) return float is separate;
      function fs (x : float) return float is separate;
      :
   end Newton;
   separate (Newton) function f (x : float) return float
      is begin
            return sin(Ø.5 * x) + (x** 3 - 1.Ø)/(x* x+ 1.Ø);
         end f;
   separate (Newton) function fs (x : float) return float
      is q,y : float;
         begin
            q:= x * x;
            y:= q + 1.Ø;
            return Ø.5*cos(Ø.5*x) - (q*q+3.Ø*q+2.Ø*x)/(y*y);
         end fs;
```

2. VEREINBARUNGEN

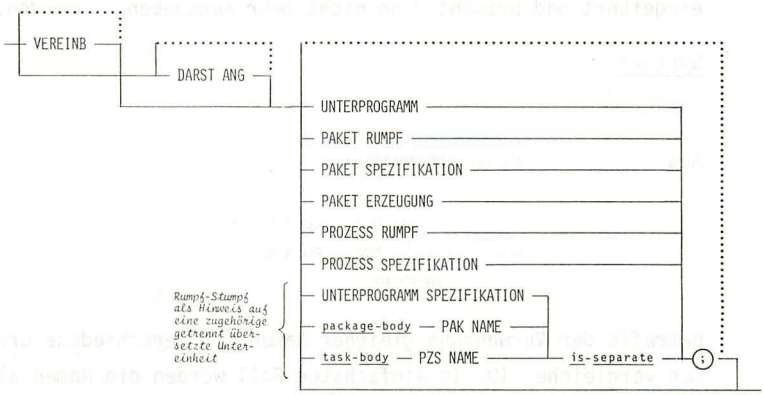
Die Namen für Typen, Konstanten und Variablen, Unterprogramme, Pakete, Prozesse und Ausnahmen müssen vor ihrer Verwendung als solche kenntlich gemacht werden, d.h. sie müssen in einer entsprechenden Vereinbarung stehen. Standardnamen dürfen nicht vereinbart werden, falls sie in ihrer ursprünglichen Bedeutung verwendet werden sollen. Die Namen von Marken, Blöcken und Schleifen werden implizit durch ihr Auftreten vereinbart.

Eine Vereinbarung für Parameter, Eingänge, Diskriminanten oder Komponenten geschieht innerhalb einer der oben angegebenen Vereinbarungen.

Wir unterscheiden zwischen einer Folge von VEREINBARUNGEN und einem VEREINBARUNGSTEIL.

Stets muß in ADA eine Größe vereinbart werden, bevor sie verwendet wird. Abgesehen von dieser Grundregel ist die Reihenfolge der Vereinbarungen beliebig. Im Vereinbarungsteil kommen möglicherweise zu den Vereinbarungen noch die Darstellungsangaben und die Rümpfe der spezifizierten Programmeinheiten oder aber Hinweise auf separat übersetzte Programmeinheiten in Form von Rumpfstümpfen. Dadurch wird nun eine etwas strengere Reihenfolge festgelegt, z.B. darf nach einer Darstellungsangabe keine Typvereinbarung mehr stehen. Die genaue Syntax zeigt das Diagramm

ADA 17: VEREINBARUNGSTEIL (VEREINBTL)



Semantik:

Zusammengehörige Teile von Programmeinheiten, also PAKETSPEZIFIKATION und PAKETRUMPF, PROZESSPEZIFIKATION und PROZESSRUMPF, getrennt vorhandene UNTERPROGRAMMSPEZIFIKATION und UNTERPROGRAMM, müssen in einem Vereinbarungsteil auftreten.

Hinweise:

(1) Man beachte, daß z.B. eine Paketspezifikation einerseits eine VEREINBARUNG ist, andererseits aber auch nach einer DARSTELLUNGSANGABE auftreten kann.

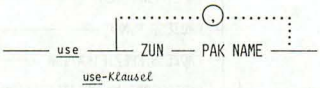
(2) Als Faustregel über die Reihenfolge im Vereinbarungsteil kann man sich merken: Zuerst Vereinbarungen, dann Darstellungsangaben, dann Rümpfe.

(3) Die einzige Stelle, an der eine Folge von Vereinbarungen, gefolgt von Darstellungsangaben, anstelle eines Vereinbarungsteils steht, ist innerhalb des geschützten Teils einer Paketvereinbarung. Es scheint auch wenig sinnvoll, Rümpfe von Programmeinheiten aus dem geschützten Teil auszulagern.

Durch Angabe einer use-Klausel wird der Zuname für die Größen des entsprechenden Paketes in der aktuellen Programmeinheit eingeführt und braucht dann nicht mehr angegeben zu werden.

Syntax:

Aus ADA 18: VEREINBARUNG



Betreffs der Verwendung gleicher Namen für verschiedene Größen vergleiche 10. Im einfachsten Fall werden die Namen aller verwendeten Größen verschieden sein.

2.1 DIE VEREINBARUNG VON TYPEN

Es gibt Typen für Datenobjekte und für Prozesse. Die Vereinbarung von Prozeßtypen wird in 2.6 behandelt. Beinhaltet das Programm eine Eingabe oder Ausgabe, so werden noch Typen für Dateien (Files) gebraucht. Diese werden in speziellen Paketen bereitgehalten. Die Handhabung von Files wird in 11 beschrieben. Ein (Daten-)Typ besteht aus einem Wertebereich und einer Menge vorgegebener Operationen.

Die Typen lassen sich in 3 Gruppen unterteilen. Die skalaren Typen, die strukturierten Typen und die Zeigertypen. Obwohl sich die Files als spezielle strukturierte Typen auffassen ließen, werden sie wegen ihrer Sonderstellung als eine eigene Gruppe betrachtet. Die weitere Aufteilung sowie die im Text und den Syntaxdiagrammen verwendeten Abkürzungen zeigt das folgende Bild.

Übersicht über die Typen

skalar	diskret Θ_D	Aufzählung	boolean	B	Θ
			character	CH	
			Aufzählung	AZ	
	numerisch		integer	I	
		reell	Gleitpunkt	G	R
			Festpunkt	F	
strukturiert		Array	Array	A	
			String	ST	
			Record	REC	
Zeiger			Zeiger	Z	
File		File	in_file	IF	Φ
			out_file	OF	
			inout_file	IOF	
		Textfile	Eingabetextfile	ITF	
			Ausgabetextfile	OTF	
Prozeß			Prozeß	PZS	

Die Datentypen werden zur Menge

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$

und die Filetypen zur Menge

$\Phi = \{IF,OF,IOF,ITF,OTF\}$

zusammengefaßt.

Ferner fassen wir die diskreten Typen in der Menge

$\Theta_D = \{B,CH,AZ,I\}$ zusammen.

Syntax:

aus ADA 18 : VEREINBARUNG

Semantik:

Zuerst wird der Typname eingeführt, dann folgt die Verarbeitung der formalen Diskriminantenliste, falls vorhanden, und anschließend die Verarbeitung der Typdefinition. Verschiedene Typvereinbarungen erzeugen stets verschiedene Typen, welche stets imkompatibel sind. Siehe 2.1.4.

Nur für Zeigertypen oder Typen, die Zeigertypen enthalten, kann die unvollständige Typvereinbarung benutzt werden, um rekursive oder gegenseitig voneinander abhängige Typen zu vereinbaren.

2.1.1 DIE SKALAREN TYPEN

Die Wertebereiche der skalaren Typen sind linear geordnet. Entsprechend dieser Ordnung lassen sich durch Einschränkung des Wertebereichs Unterbereiche angeben.

2.1.1.1 DIE DISKRETEN TYPEN

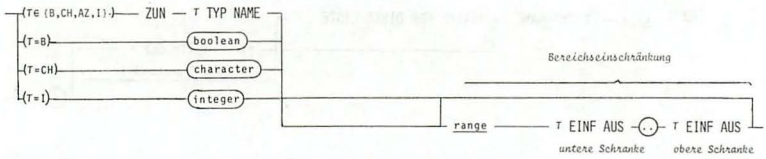
Die diskreten Typen treten auf als Indexbereiche von Feldern, als Laufbereiche in Wiederholungsanweisungen, als Diskriminanten und zur Auswahl in Auswahlanweisungen.

Θ_D = {B,CH,AZ,I} sei die Menge der diskreten Typen.

Syntax:

Aus

ADA 22: T TYP



Semantik:

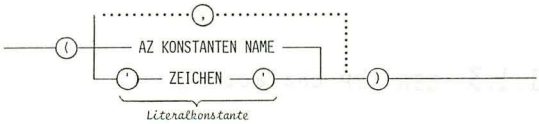
Jedem Wert eines diskreten Typs ist eine ganzzahlige Positionsnummer zugeordnet, die die Ordnung bestimmt. Sei λ der Wert der unteren Schranke, ρ der Wert der oberen Schranke. Dann erfüllt ein Wert γ die Bereichseinschränkung, falls $\lambda \leq \gamma \leq \rho$. Falls $\lambda > \rho$, so ist der diskrete Unterbereich leer. Eine Bereichseinschränkung (λ .. ρ) ist verträglich mit einer früheren Bereichseinschränkung (λ' .. ρ'), falls gilt $\lambda' \leq \lambda$ und $\rho \leq \rho'$ oder falls $\lambda > \rho$.

2.1.1.1.1 DER AUFZÄHLUNGSTYP

Durch Aufzählung von frei wählbaren Namen oder Zeichenkonstanten kann ein Wertebereich und damit ein Typ definiert werden.

Syntax:

ADA 20: AZ TYP DEFINITION (AZ TYP DEF)



Semantik:

Die aufgezählten AZ KONSTANTEn, das sind AZ KONSTANTENNAMEN oder Zeichenkonstanten, bilden einen Wertebereich; sie werden durch ihr Auftreten in der Aufzählung definiert und sind gemäß den Regeln der Blockstruktur wie definierte Größen zu behandeln. Eine AZ Konstante kann mehrfach belegt sein, d.h. in mehreren AZ Typdefinitionen auftreten. Die AZ Konstanten sind von links nach rechts durchnumeriert. Die Positionsnummer der ersten Konstanten ist 0.

Die Typen boolean und character sind spezielle vordefinierte Aufzählungstypen. Da für sie mehr Operationen als für normale Aufzählungstypen bekannt sind, sind sie allerdings eigenständige Typen.

Die Konstanten des AZ Typs sind in ADA 38 angegeben.

Beispiel:

```
type ehestand is (ledig,verheiratet,geschieden,verwitwet);
type wochentag is (mo,di,mi,do,fr,sa,so);
type tonart is (do,re,mi,fa,so,la,si);
```

2.1.1.1.2 DER TYP boolean

```
type boolean is (FALSE,TRUE);
```

Die Werte aller Vergleiche sind boolean. Boolesche Ausdrücke dienen zur Verzweigung in bedingten Anweisungen.

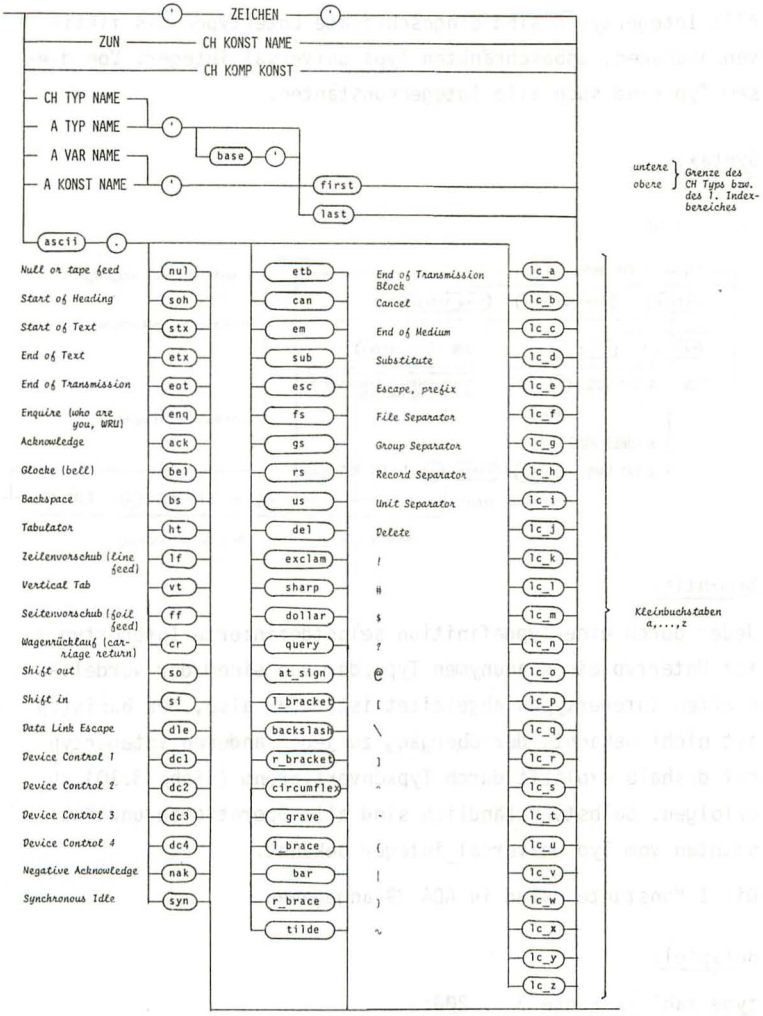
Die Konstanten sind in ADA 36 angegeben.

2.1.1.1.3 DER TYP character

Der Typ character besteht aus allen 128 ASCII-Zeichen. Dabei gibt es (im Paket standard) für die nicht druckbaren oder nicht zum Zeichensatz gehörigen Zeichen vorgeschriebene Konstantennamen.

Die Konstanten sind in ADA 37 angegeben:

ADA 37: CH KONSTANTE (CH KONST)



2.1.1.1.4 DER TYP integer

Syntax:

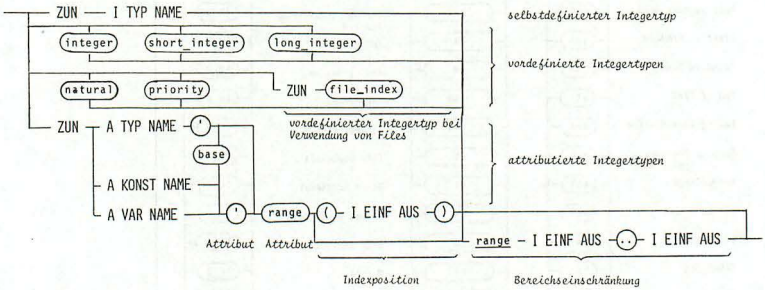
ADA 20: I TYP DEFINITION (I TYP DEF)

Semantik:

Alle Integertypen sind eingeschränkte Untertypen des fiktiven, anonymen, unbeschränkten Typs universal_integer. Von diesem Typ sind auch alle Integerkonstanten.

Syntax:

ADA 22: I TYP



Semantik:

Jeder durch eine Typdefinition selbstdefinierte Integertyp ist Untertyp eines anonymen Typs, der von einem der vordefinierten Integertypen abgeleitet ist, d.h. also, der Basistyp ist nicht bekannt, der Übergang zu jedem anderen Integertyp hat deshalb explizit durch Typkonvertierung (siehe 3.10) zu erfolgen. Selbstverständlich sind alle Operationen und Konstanten vom Typ universal_integer bekannt.

Die I Konstanten sind in ADA 39 angegeben.

Beispiel:

type zahl is range 3 .. 200;
    ist äquivalent zu
subtype zahl is new short_integer range 3 .. 200;
    oder, falls der Typ short_integer nicht implementiert ist oder nicht bis 200 reicht, äquivalent zu

subtype zahl is $\underbrace{\text{new integer}}_{\text{anonymer Typ}}$ range 3 .. 200;

Sei α ein eingeschränkter Arraytyp, Arrayvariable oder -konstante, κ der Wert der Indexposition, so bezeichnet α'range(κ) den Untertyp des κ-ten Indexbereiches von α. Fehlt die Angabe der Indexposition, so ist automatisch $\kappa := 1$ gesetzt. α'base'range(κ) bezeichnet den Basistyp des κ-ten Indexbereichs. Der Wert der Indexposition muß zur Übersetzungszeit berechenbar sein.

Die vordefinierten Typen integer, short_integer und long_integer sind symmetrisch zu Null (bis auf eine zusätzliche negative Zahl im Falle von b-1 Komplementdarstellung). Ihre Wertebereiche unterscheiden sich deutlich. Nur der Typ integer ist für jede Implementierung vorgeschrieben. Der Typ natural ist der Untertyp aller positiven integer-Zahlen.

subtype natural is integer range 1 .. integer'last;

Der Typ priority ist ein implementierungsbedingter Untertyp von integer für die Priorität von Prozessen.

Der implementierungsbedingte Typ file_index dient zur Angabe von Filepositionsnummern.

Hinweise:

(1) Die integer Typen gehören sowohl zu den skalaren als auch zu den numerischen Typen. Man kann sie also zur Indizierung gebrauchen und zu den reellen Typen konvertieren. Auch Typkonversionen zwischen den verschiedenen vordefinierten integer Typen sind möglich. Eine Konvertierung vom Typ universal_integer geschieht dagegen automatisch.

(2) Die vordefinierten Integertypen bezeichnen wir in den Diagrammen mit folgenden Abkürzungen

integer short_integer, long_integer	I
natural	N
priority	P
file_index	FI

2.1.1.2 REELLE TYPEN

Der Wertebereich eines reellen Typs ist eine maschinenunabhängige Teilmenge der reellen Zahlen: die sogenannten Modellzahlen. Jede Implementierung muß mindestens die Modellzahlen exakt darstellen. Die Rundung $\square$ einer reellen Zahl (im ADA-Sprachgebrauch eines Wertes vom fiktiven anonymen Typ universal_real) in die Menge der Modellzahlen M hat folgende Eigenschaften

(R1) $\bigwedge_{x \in M} \square x = x$, d.h. Modellzahlen verändern sich durch die Rundung nicht, sie werden exakt dargestellt.

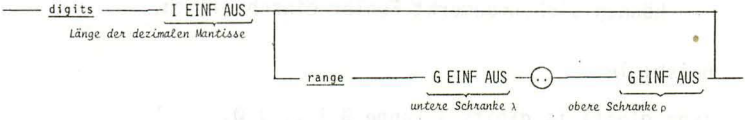
(R2) Seien y und z die beiden unmittelbar benachbarten Modellzahlen von x mit $y < z$.

$\bigwedge_{x \in [y,z]} \square x \in \{y,z\}$, d.h. durch die Rundung werden alle reellen Werte aus einem Modellintervall [y,z] auf eine der Grenzen abgebildet.

2.1.1.2.1 DER GLEITPUNKTTYP

Syntax:

ADA 20: G TYP DEFINITION (G TYP DEF)



Semantik:

Sei δ der Wert der Länge der dezimalen Mantisse. Dann ist die Modellmenge das Gleitpunktsystem $M = G(\beta,2,-4\beta,4\beta)$, d.h. die Menge aller Gleitpunktzahlen zur Basis 2 mit $\beta \geq \delta * \ln(10)/\ln(2)$ Mantissenziffern und dem Exponentenbereich -4β bis 4β. (Die Werte der Länge der dezimalen Mantisse und der unteren und oberen Schranke) δ,λ und ρ müssen zur Übersetzungszeit berechnet werden können.

Hinweise:

(1) Gleitpunktzahlen approximieren die reellen Zahlen mit relativer Genauigkeit, d.h. $\bigwedge_{x\in[\lambda,\rho]} \left|\frac{x-\square x}{x}\right| < 2^{1-\beta}$.

(2) Alle Modellzahlen sind von der Form $\mu\cdot 2^{\eta}$.
Dabei ist μ der Wert der dualen Mantisse und η der Wert des ganzzahligen Exponenten. Es gilt
$0.5 \leq \mu < 1$,
$-4\beta \leq \eta \leq 4\beta$,
μ hat β Stellen nach dem Punkt

(3) Durch β, d.h. durch δ wird also die Genauigkeit der Approximation beschrieben.

(4) Da das Dualsystem zur Beschreibung der maschinenunabhängigen Modellmenge verwendet wird, in der Darstellung von

Konstanten im Programm jedoch meist das gewohnte Dezimalsystem benutzt werden wird (s. 3.5.1), welches ja auch bei der Typdefinition bevorzugt wird, ist große Vorsicht bei Genauigkeitsfragen geboten. Denn durch automatische Konvertierung, die auch bei Dezimalmaschinen auftritt (!), können sich unbemerkt Fehler einschleichen.

Beispiele:

type gleit1 is digits 1 range Ø.1 .. 1.Ø;

beschreibt die Dezimalzahlen zwischen Ø.1 und 1 mit einer Mantissenziffer:

Ø.1, Ø.2, Ø.3, Ø.4, Ø.5, Ø.6, Ø.7, Ø.8, Ø.9, 1.Ø

In der Menge der zugehörigen Modellzahlen

$M = G(4,2,-8,8) \cap [Ø.1,1]$

liegen nur Ø.5 und 1, während alle anderen Werte gerundet werden müssen, z.B.

$\square$ $Ø.3 = Ø.1ØØ1_2\text{-}1 = Ø.28125$

oder $\square$ $Ø.3 = Ø.1Ø1Ø_2\text{-}1 = Ø.3125$

$Ø.11ØØ_2\text{-}3 = Ø.Ø9375$ gehört zum Wertebereich des Typs, falls $\square$ $Ø.1 = Ø.Ø9375$ ist!

Syntax:

ADA 22: G TYP

Semantik:

Jeder durch eine Typdefinition selbstdefinierte Gleitpunkttyp ist Untertyp eines anonymen Typs, der von einem der vordefinierten Gleitpunkttypen abgeleitet ist. Der Basistyp ist also unbekannt und kann je nach Implementierung ein anderer sein.

(Die *Genauigkeits-* und *Bereichseinschränkungen* bei Untertyp- oder Variablenvereinbarung müssen mit der des Basistyps verträglich sein.) Sei δ die Länge der dezimalen Mantisse, λ der Wert der unteren, ρ der Wert der oberen Schranke. Die entsprechenden Größen des Basistyps seien δ', λ', ρ'. Dann heißt die Genauigkeitseinschränkung des Untertyps v e r t r ä g l i c h mit der des Basistyps, falls $\delta < \delta'$, die Bereichseinschränkung heißt v e r t r ä g l i c h , falls $\lambda,\rho \in [\lambda',\rho']$.

Jede Implementierung kennt den Typ float. Die Typen long_float und short_float unterscheiden sich, falls vorhanden, deutlich in der Genauigkeitseinschränkung von diesem.

Die Werte δ,λ und ρ bei Untertyp- oder Variablenvereinbarung brauchen erst zur Laufzeit bekannt zu sein.

Die G KONSTANTEN sind in ADA 40 angegeben.

Hinweis:

Durch eine Genauigkeitseinschränkung bei Untertypen wird eine geringere Genauigkeit verlangt als bei dem Grundtyp. Dadurch kann die Verarbeitungsgeschwindigkeit erhöht werden. Man kann nicht davon ausgehen, daß für jede Modellmenge eine eigene Arithmetik implementiert ist, daß also eine Beschleunigung in jedem Fall eintritt.

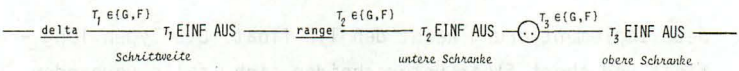
Beispiel:

type real is digits 10;
ist äquivalent zu
subtype real is new float digits 10;
falls die Genauigkeit von float 10 Dezimalziffern in der Mantisse zuläßt, sonst
subtype real is new long_float digits 10;

2.1.1.2.2 DER FESTPUNKTTYP

Syntax:

ADA 20: F TYP DEFINITION (F TYP DEF)



Semantik:

Der Wert δ der *Schrittweite* muß positiv sein und zur Übersetzungszeit berechenbar (statischer Ausdruck).

Die Modellmenge des Typs besteht aus aufeinanderfolgenden ganzzahligen Vielfachen einer implementierungsabhängigen Zahl $\Delta \leq \delta$. Sie ist symmetrisch zu Null, ungeachtet der Bereichseinschränkung. Der Abstand der Schranken zur nächsten Modellzahl ist höchstens gleich δ.

Hinweise:

(1) Festpunktzahlen approximieren die reellen Zahlen mit absoluter Genauigkeit, d.h. $\bigwedge_{x \in [\lambda,\rho]} |x - \square x| \leq \Delta$.

(2) Δ sollte bei Dualmaschinen eine Potenz von 2 sein. Sei

$\alpha = -2^N + 1$, $\beta = 2^N - 1$, dann besteht die Modellzahlen-

menge aus den Zahlen $i * \Delta$, wobei i die ganzen Zahlen von α bis β durchläuft. N ist dabei so gewählt, daß zwei ganze Zahlen $\nu, \mu \in [\alpha, \beta]$ existieren, so daß gilt:

$$|\lambda - \nu \cdot \Delta| \leq \delta \quad \text{und} \quad |\rho - \mu \cdot \Delta| \leq \delta.$$

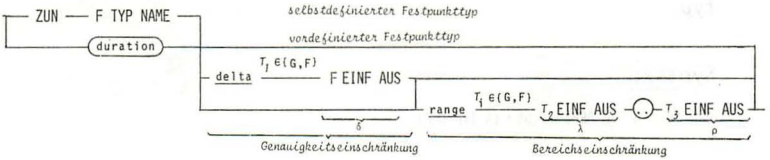
Beispiel:

type fix1 is delta Ø.1 range Ø.1 .. 1.25;
beschreibt die Dezimalzahlen Ø.1, Ø.2, ...,1.Ø, 1.1, 1.2 .
Die Menge der Modellzahlen bei dualer Maschine ist für diesen Bereich also

$$M = \{i \cdot 2^{-4} | i = -2^{-5} + 1 \ (1) \ 2^{-5} - 1\}$$
$$= \{Ø.125, Ø.1875, Ø.25, Ø.3125, Ø.375, Ø.4375, Ø.5, Ø.5625, Ø.625, Ø.6875, Ø.75, Ø.8125, Ø.875, Ø.9375, 1, 1.Ø625, 1.125, 1.1875, 1.25\}.$$

Syntax:

ADA 22: F TYP



Semantik:

Eine Genauigkeitseinschränkung (δ) heißt verträglich mit einer früheren (δ'), falls $\delta \geq \delta'$, eine Bereichseinschränkung (λ, ρ) heißt verträglich mit einer früheren (λ', ρ'), falls $\lambda, \rho \in [\lambda', \rho']$. Der vordefinierte Typ duration (D) dient zur Zeitangabe (in sec.) in Verzögerungsanweisungen (s. 9.4).
Die F Konstanten sind in ADA 40 angegeben.

2.1.2 STRUKTURIERTE TYPEN

Die Zusammenfassung von Werten aus beliebigen Datentypen kann aufgefaßt werden als Wert eines neuen Datentyps, der dann aus allen Werten besteht, die durch gleichartige Zusammenfassung entstanden sind. Die Art der Zusammenfassung bezeichnet man als Strukturierung, und dementsprechend heißt auch der Datentyp strukturiert. Es gibt in ADA 2 Formen strukturierter Datentypen

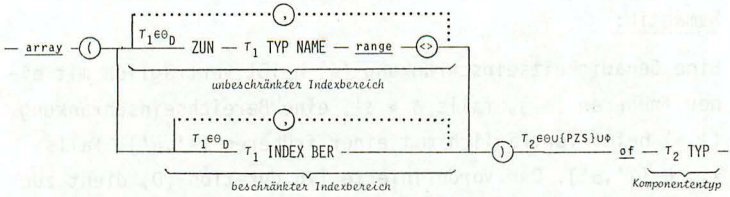
- Felder oder Arrays
- Verbunde oder Records

2.1.2.1 DER ARRAY TYP

Ein Feld (auch Array genannt) besteht aus einer festen Anzahl von Komponenten gleichen Typs. Die einzelnen Komponenten werden durch Indizes gekennzeichnet. Die Typdefinition eines Feldes enthält also Indexbereiche und den Komponententyp.

Syntax:

ADA 20: A TYP DEFINITION (A TYP DEF)



Als Basistypen für Indexbereiche kommen alle diskreten Typen in Betracht.

In einer Typdefinition müssen die Indexbereiche entweder

alle unbeschränkt oder alle beschränkt sein.

Semantik:

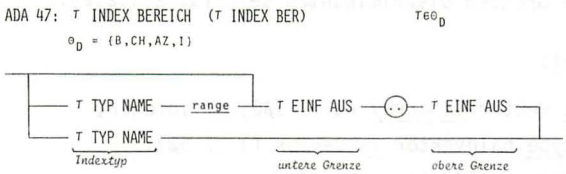
Ein Feld oder Array besteht aus einer durch die Indexbereiche festgelegten Anzahl von Komponenten des gleichen Typs. Mehrdimensionale Arrays können als Arrays von Arrays aufgefaßt werden, jedoch hat die Indizierung (anders als in PASCAL) gemäß der Typdefinition zu erfolgen.

Nur in einer Typvereinbarung darf eine Array Typdefinition mit unbeschränkten Indexbereichen stehen.

Korrekte Komponentenzugriffe sind

z.B. A: array (1 .. 10) of array (1 .. 10) of integer;
B: array (1 .. 10, 1 .. 10) of integer;
A(1)(3), aber B(1,3) und nicht B(1)(3)

Syntax:



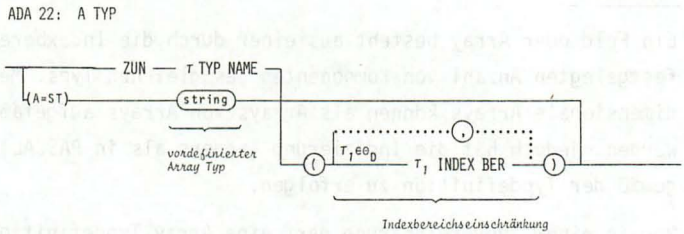
Semantik:

Ist ein Indexbereich leer (s. 2.1.1.1), so hat das Feld keine Komponenten.

Fehlt für T = I die Angabe des Typnamens, so wird integer als Basistyp genommen. Der Wert der Grenzen braucht erst zur Laufzeit berechnet werden zu können. Man spricht dann von dynamischen Arrays.

Der Wert der Grenzen muß innerhalb des Wertebereichs des Indextyps liegen.

Syntax:



Semantik:

Eine Indexbereichseinschränkung betrifft stets alle Indizes. Ein Array Typ mit beschränktem Indexbereich darf keine weitere Indexbereichseinschränkung erfahren.

In einer Variablenvereinbarung muß ein Array Typ mit beschränktem (möglicherweise dynamischem) Indexbereich angegeben werden. Treten dynamische Arrays in Records auf, so müssen die Grenzen Diskriminanten sein (s. 2.1.2.2).

Beispiel:

```
type vektor is array (1 .. 100) of integer;
subtype halbvektor is vektor (1 .. 50);
   ist unzulässig

type v is array (integer range <>) of integer;
subtype vektor is v (1 .. 100);
subtype halbvektor is v (1 .. 50);
   ist korrekt
```

2.1.2.1.1 DER STRING TYP

Der String Typ ist ein vordefinierter eindimensionaler unbeschränkter Array Typ mit Komponententyp character.

type string is array (natural range <>) of character;

Da für diesen Typ spezielle Operationen vorgesehen sind und er als Ergebnistyp attributierter Funktionen vorkommt, wird er an manchen Stellen besonders erwähnt.

Die ST Konstanten sind in ADA 42 angegeben.

2.1.2.2 DER RECORD TYP

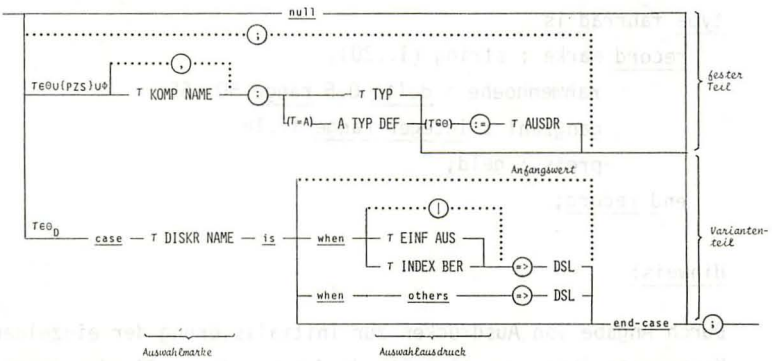
Ein Record (Verbund) besteht aus endlich vielen Komponenten, die von verschiedenem Typ sein können. Die Komponenten werden durch ihren Namen angesprochen.

Syntax:

ADA 20: REC TYP DEFINITION (REC TYP DEF)

ADA 21: DATENSATZLISTE (DSL)

θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} θ_D = {IF,OF,IOF,ITF,OTF} θ_D = {B,CH,AZ,I}



Semantik:

Ist die Datensatzliste null, so hat das Record keine Komponenten.

Hat die Datensatzliste in einer RECORD TYP DEFINITION keinen

Variantenteil, so haben alle Objekte dieses Typs die gleichen Komponenten.

In der A TYP DEF dürfen nur eingeschränkte Indexbereiche vorkommen, die Grenzen müssen entweder zur Übersetzungszeit berechnet werden können (statische Arrays) oder formale Diskriminanten des Record Typs sein (dynamische Arrays). Es dürfen jedoch keine Ausdrücke mit den formalen Diskriminanten gebildet werden.

Die Datensatzliste a1,a2,a3: a; ist äquivalent zu der Datensatzliste a1: a; a2: a; a3: a; Die Auswahlmarke ist eine formale Diskriminante. Die statischen Auswahlausdrücke müssen jeden Wert des entsprechenden Diskriminantentyps genau einmal annehmen, oder ein others-Zweig muß vorhanden sein.

Beispiele:

```
type geld is
   record mark : integer; pfennig : range 0..99;
   end record;
type fahrrad is
   record marke : string (1..20);
          rahmenhoehe : delta 0.5 range 50..65;
          gangzahl : integer range 1..18;
          preis : geld;
   end record;
```

Hinweis:

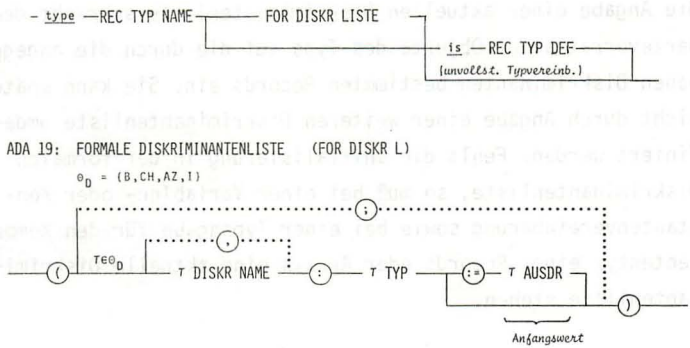
Durch Angabe von Ausdrücken zur Initialisierung der einzelnen Komponenten kann man eine Standardvorbesetzung für den gesamten Recordtyp erreichen.

Wie in PASCAL tritt maximal ein Variantenteil am Schluß auf. Dieser kann selbstverständlich wieder Variantenteile enthalten, usw.

Anders als in PASCAL muß die Auswahlmarke einen Namen haben. Zur Auswahl der Varianten können auch Bereiche herangezogen werden. Alle Werte des Diskriminantentyps müssen angenommen werden. Notfalls können nicht gebrauchte mit <u>others</u> <u>null</u>; zusammengefaßt werden.

Eine Recordtypvereinbarung kann eine formale Diskriminantenliste beinhalten.

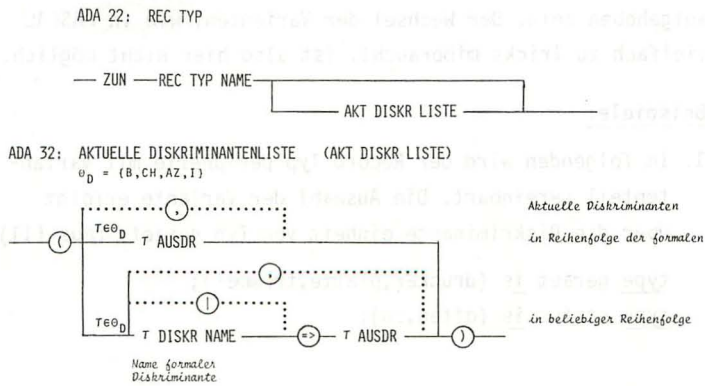
<u>Syntax:</u> REC TYP VEREINBARUNG aus ADA 18 VEREINBARUNG



<u>Semantik:</u>

Entweder werden alle Diskriminanten initialisiert oder keine.

<u>Syntax:</u>



Semantik:

In der aktuellen Diskriminantenliste wird jeder *formalen Diskriminante* ein Wert des entsprechenden Typs zugewiesen. Es gibt zwei Möglichkeiten der Ersetzung; entweder werden die *aktuellen Diskriminanten* der *Reihenfolge* nach zugewiesen, oder es werden die Namen der formalen Diskriminanten wiederholt (*beliebige Reihenfolge*). Diese beiden Arten können gemischt werden, dann muß jedoch die Ersetzung nach Namen am Schluß stehen.

Die Angabe einer aktuellen Diskriminantenliste schränkt den Wertevorrat aller Objekte des Typs auf die durch die angegebenen Diskriminanten bestimmten Records ein. Sie kann später nicht durch Angabe einer weiteren Diskriminantenliste umdefiniert werden. Fehlt die Initialisierung in der formalen Diskriminantenliste, so muß bei einer Variablen- oder Konstantenvereinbarung sowie bei einer Typangabe für den Komponententyp eines Records oder Arrays eine aktuelle Diskriminantenliste stehen.

Hinweis:

Durch eine Diskriminantenliste läßt sich ein Record Typ parametrisieren. Bis zur Variablenvereinbarung muß diese Parametrisierung allerdings durch Angabe aktueller Diskriminanten aufgehoben sein. Der Wechsel der Varianten, wie in PASCAL vielfach zu Tricks mißbraucht, ist also hier nicht möglich.

Beispiele:

1. Im folgenden wird der Record Typ peripherie mit Variantenteil vereinbart. Die Auswahl der Variante erfolgt über die Diskriminante einheit vom Typ geraet: (aus [1])

   <u>type</u> geraet <u>is</u> (drucker,platte,trommel);
   <u>type</u> status <u>is</u> (offen,zu);

```
type peripherie (einheit : geraet := platte) is
   record
      zustand : status;
      case einheit is
         when drucker =>
            zeilenzahl : integer range 1..seitenlaenge;
         when others =>
            zylinder : index;
            spur : nummer;
      end case;
   end record;
```

Die Vereinbarung

```
   subtype druck_peripherie is peripherie (drucker);
```

kann auch durch

```
   type druck_peripherie is
      record
         zustand : status;
         zeilenzahl : integer range 1 .. seitenlaenge;
      end record;
```

dargestellt werden.

2. Im folgenden wird der Record Typ fuehrerschein mit Varianteneil vereinbart. Die Auswahl der Variante erfolgt über die Diskriminante amtliches vom Typ eintragungen:

```
type name is
   record vorname : string (1..10);
          zweiter_vorname : character;
          nachname : string (1..10);
   end record;
type klasse is (IA,IB,II,III,IV);
type monat_name is
      (jan,feb,mae,apr,mai,jun,jul,aug,sep,okt,nov,dez);
type datum is
```

```
    record tag : integer range 1..31;
           monat : monat_name;
           jahr : integer;
    end record;
type eintragungen is
    (normal,eingeschraenkt,spezial_erlaubnis,vip);
type fuehrerschein (amtliches : eintragungen) is
    record fahrer : name;
           fuehrerschein_klasse : klasse;
           fuehrerschein_nummer : integer;
           ausstellungs_datum : datum;
           case amtliches is
              when spezial_erlaubnis =>
                 fahrzeug_typ : integer;
                 passagiere : boolean;
                 zonen_nummer : integer;
              when eingeschraenkt =>
                 kontaktlinsen : boolean;
                 nicht_bei_dunkelheit : boolean;
                 nur_automatik_getriebe : boolean;
              when others => null;
           end case;
        end record;
```

2.1.3 DER ZEIGERTYP

Unter den bisher behandelten Datentypen vermissen wir zwei wesentliche Eigenschaften:

(1) Eine Variable sollte während der Ausführung eines Blokkes, falls sie benötigt wird, erzeugt werden können.

(2) Eine unbestimmte Anzahl von Elementen, z.B. eine Kartei,

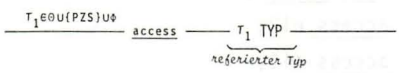
sollte auf einfache Weise bearbeitet werden können. Insbesondere sollte das Anfügen, Einfügen, Herausnehmen (Löschen) von Elementen leicht möglich sein.

Diese Lücke füllt die Zeigervariable. Dabei ist eine Zeigervariable z ein Zeiger (Verweis, Zugriff), der auf eine Variable z.all des sogenannten referierten Typs verweist. Diese referierte Variable wird erst während der Ausführung einer Programmeinheit durch die Speicherbelegungsfunktion (Allokator) new generiert (dynamische Variable), während die Zeigervariable selbst wie Variablen aller anderen Typen vereinbart werden muß (statische Variable). Damit kann man nahezu alle erwünschten Datenstrukturen wie lineare Listen, Bäume, ... usw. erzeugen.

Ein Zeiger, der auf eine Variable eines bestimmten Typs verweist, ist vorstellbar als Adresse (Speicherplatznummer) dieser Variablen. Der Typ dieser Variablen, der sogenannte *referierte Typ*, ist bei der Zeigertypdefinition anzugeben.

Syntax:

ADA 20: Z TYP DEFINITION (Z TYP DEF)



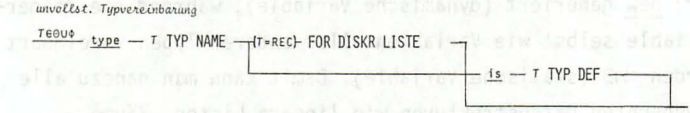
Semantik:

Der Wertebereich eines Zeigertyps ist die Menge der Verweise auf Variable vom referierten Typ, erweitert um den Wert null, einem konstanten Zeiger, der zu jedem Zeigertyp gehört und auf keine Variable verweist. null ist der Standardanfangswert aller Zeigervariablen.

Für Zeigertypen und strukturierte Typen, die Zeigertypen enthalten, kann die unvollständige Typvereinbarung auftreten. Sie ist zu verwenden, falls der referierte Typ Komponenten

des vereinbarten Zeigertyps enthält. Mit ihrer Hilfe lassen sich also rekursive und gegenseitig abhängige Datenstrukturen formulieren unter der Bedingung, daß jede Größe bekannt sein muß, falls sie verwendet wird. Eine vollständige Typvereinbarung eines unvollständig vereinbarten Typs hat noch im gleichen Vereinbarungsteil zu folgen.

Aus ADA 18: VEREINBARUNG

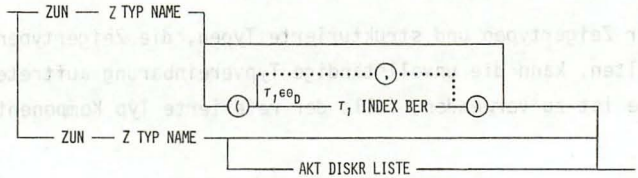


Beispiel: Unvollständige Typvereinbarung

```
type zeigerpaar;
type el1 is record i : integer;
                   p : zeigerpaar;
            end record;
type p2;
type el2 is record i : character;
                   p: p2;
            end record;
type p2 is access el2;
type p1 is access el1;
type zeigerpaar is record z1 : p1;
                          z2 : p2;
                end record;
```

Syntax:

ADA 22: Z TYP



Semantik:

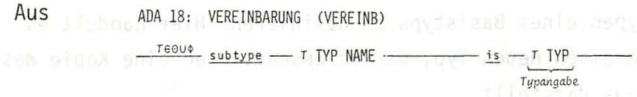
Die Indexeinschränkung bzw. aktuelle Diskriminantenliste bezieht sich auf den referierten Typ. Sie muß spätestens bei der Generierung der dynamischen Variablen durch die Speicherbelegungsfunktion new angegeben werden (vgl. Array- bzw. Recordtyp).

2.1.4 UNTERTYPEN

Es gibt in ADA die Möglichkeit, Untertypen zu vorliegenden Typen, die dann als Basistypen bezeichnet werden, zu definieren. Das wird in der Regel dazu benutzt, Wertebereiche einzuschränken oder die Genauigkeit bei reellen Typen herabzusetzen.

Syntax:

Aus ADA 18: VEREINBARUNG (VEREINB)



Semantik:

Enthält die *Typangabe* keine Einschränkung, so wird ein neuer, weiterer Name für den Basistyp (Muttertyp) eingeführt. Treten Einschränkungen des Basistyps auf, so müssen sie mit den bereits bestehenden Einschränkungen verträglich sein.

Basistyp und Untertyp sind k o m p a t i b e l , d.h. es können die Werte der beiden Typen ohne Konversion verwendet werden.

Beispiele:

subtype temperatur is float digits 4 range -275.0..1.0e9;

subtype badetemperatur is temperatur range 15.0..48.0;

Der Typ badetemperatur ist mit dem Typ temperatur kompatibel. Durch

```
subtype taxi_fuehrerschein is
        fuehrerschein (amtliches => spezial_erlaubnis);
```

wird ein Untertyp des in 2.1.2.2 beschriebenen Record Typs "fuehrerschein" vereinbart.

Hinweis:

Es können Untertypen von Untertypen vereinbart werden.

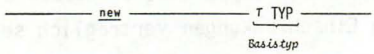
Wir unterscheiden im folgenden nicht zwischen Basistyp und Untertyp, falls dies nicht ausdrücklich betont wird.

2.1.5 ABGELEITETE TYPEN

Neben den Untertypen gibt es noch die Möglichkeit, abgeleitete Typen eines Basistyps zu definieren. Hier handelt es sich um einen neuen Typ, der im wesentlichen eine Kopie des Basistyps darstellt.

Syntax:

Aus ADA 20: τ TYP DEFINITION (τ TYP DEF) $\tau \in \Theta \cup \Phi$



Semantik:

Es wird ein neuer Typ der gleichen Typklasse eingeführt (deshalb das gleiche Symbol τ), für den bis auf eventuelle Einschränkungen des Wertebereichs oder Lockerungen der Genauigkeitsforderungen bei reellen Typen folgendes gilt:

- Der Wertebereich ist eine Kopie des Wertebereichs des Basistyps.
- Die vordefinierten Anfangswerte des Basistyps werden übernommen, falls vorhanden.
- Basistyp und abgeleiteter Typ sind inkompatibel, jedoch besteht in beiden Richtungen die Möglichkeit der Typkonversion.
- Die Schreibweise aller Konstanten und Komponenten ist die gleiche.
- Alle Attribute und Darstellungsangaben werden übernommen.
- Bei Zeigertypen besitzen der Basistyp und der abgeleitete Typ dieselben Wertebereiche.
- Alle Operatoren, Funktionen und Prozeduren, die für den Basistyp vereinbart sind, oder in deren Argumentliste der Basistyp auftritt, stehen als abgeleitete Operatoren, Funktionen oder Prozeduren auch für den abgeleiteten Typ zur Verfügung.

Der Aufruf eines solchen abgeleiteten Unterprogramms (Operator, Funktion oder Prozedur) geschieht durch implizite Konversion der Eingabeargumente, Aufruf des Unterprogramms für den Basistyp und implizite Konversion der Ausgabeargumente. Die abgeleiteten Unterprogramme können für den abgeleiteten Typ neu definiert werden.

Hinweise:

(1) Es können abgeleitete Typen von abgeleiteten Typen gebildet werden.

(2) Für die Standardtypen und die vorgegebenen Operatoren gelten die gleichen Regeln.

(3) Die abgeleitete Typdefinition ermöglicht es, verschiedene Operatoren für den "gleichen" Typ zu vereinbaren, z.B. Addition von Gleitpunktzahlen mit Rundung nach oben oder nach unten.

Beispiel: Kompatibilität von Typen

```
type z1 is range -100 .. 100;
type z2 is new integer range -100 .. 100;
type z3 is new z1;
subtype z4 is z1;
```

z1, z2, z3 sind inkompatibel, z1 und z4 sind zwei Namen für den selben Typ. Der Übergang von z1 nach z2 bzw. nach z3 kann mittels Konversion erfolgen.

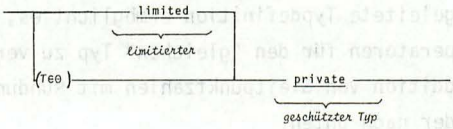
2.1.6 GESCHÜTZTE TYPEN

In einem Paket sind Datentypen und Operationen zusammengefaßt, die an anderen Stellen des Programms benutzt werden. Dabei ist die innere Struktur der verwendeten Datentypen oft nicht interessant. Zum Beispiel ist es unerheblich, ob eine komplexe Zahl mit Hilfe eines Arrays oder eines Records dargestellt wird, oder wie eine Liste aufgebaut ist, für die die benötigten Operationen "einfügen", "suchen" und "löschen" bekannt sind. Mit der Definition von geschützten Typen kann man die Information über die innere Struktur verbergen. Besonders wichtig ist das Geheimhalten von Informationen bei der Erstellung und Handhabung von Datenbanken, etwa Personalkarteien, wo dem außenstehenden Benutzer nur gewisse Teile der persönlichen Daten zugänglich sein dürfen.

Syntax:

Aus

ADA 20: T TYP DEFINITION (T TYP DEF) T E O U Φ



Semantik:

Die Vereinbarung von geschützten Typen kann nur im sichtbaren Teil einer Paketvereinbarung auftreten. Es wird nur der Name des geschützten Typs eingeführt. Die eigentliche Definition des Typs muß in einer Typvereinbarung im geschützten Teil der Paketvereinbarung nachgeholt werden. Beeinhaltet die Typvereinbarung eines geschützten Typs eine formale Diskriminantenliste, so muß auch die entsprechende Vereinbarung im geschützten Teil eine bis auf die Schreibweise der Diskriminantennamen identische formale Diskriminantenliste enthalten.

Für einen geschützten Typ stehen nur die Zuweisung sowie die Vergleichsoperatoren = und /= zur Verfügung, für einen limitierten geschützten Typ noch nicht einmal diese. Deshalb dürfen für limitierte geschützte Typen außerhalb des Paketes, in dem sie definiert werden,

- keine Konstanten vereinbart werden,
- keine Initialisierung von Variablen oder Vorbesetzung von Parametern durchgeführt werden.

Beispiele:

1.
```
package komplexe_arithmetik is
   type complex is private;

   --- Spezifikation komplexer Operationen und
   --- Funktionen

private type complex is record re,im : float;
                         end record;
```

In diesem Paket wird der Typ complex als geschützter Typ vereinbart. Damit kann man außerhalb des Paketes komplexe_arithmetik z.B. nicht auf Real- oder Imaginärteil von Objekten mit dem Typ complex zugreifen.

2. package optimale_arithmetik is
   type dot_precision is limited private;
   ⋮
   private type dot_precision is ...

   Der Typ dot_precision ermöglicht die exakte Berechnung von Skalarprodukten. Außerhalb des Paketes ist z.B. keine Zuweisung für diesen Typ möglich.

Hinweis:

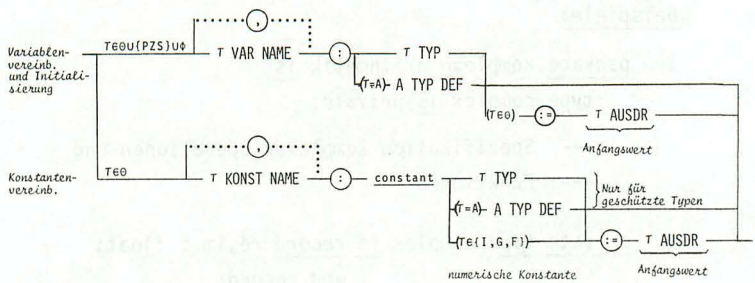
Ein Prozeßtyp kann als limitierter geschützter Typ vereinbart werden.

2.2 DIE VEREINBARUNG VON VARIABLEN UND KONSTANTEN

Syntax:

Aus

ADA 18: VEREINBARUNG (VEREINB)

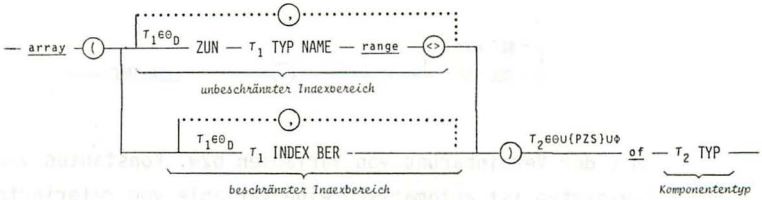


Semantik:

- Es werden die Namen der Variablen oder Konstanten eingeführt, anschließend wird der Typ oder die Arraytypdefinition ausgewertet. Dann wird der *Anfangswert* berechnet und der Variablen oder Konstanten zugewiesen.

- Fehlt die Initialisierung und gibt es für den Typ keine Standardinitialisierung, so ist der Wert der Variablen unbestimmt.
- Die Anfangswerte müssen zur Laufzeit berechnet werden können. Die Anfangswerte für numerische Konstanten, also Konstanten ohne Typangaben, dürfen nur konstante Zahlen und für diese vorhandene vordefinierte Operationen verwenden. Der Typ ist entweder universal integer bzw. universal_real, d.h. eine implizite Konvertierung nach integer bzw. Gleit- oder Festpunkt ist gewährleistet.
- Der Wert einer Konstante darf im Programm nicht verändert werden.
- Variable vom Array Typ müssen beschränkte Indexbereiche haben. Entweder muß die Indexeinschränkung an den Typnamen eines unbeschränkten Arrays angefügt werden oder, falls der Array Typ explizit definiert wird, dürfen nur beschränkte Indexbereiche auftreten.

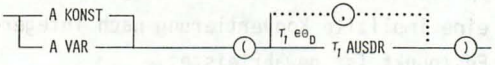
ADA 20: A TYP DEFINITION (A TYP DEF)

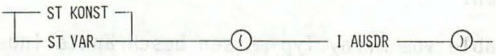


- Für Konstante vom geschützten Typ hat die Initialisierung im geschützten Teil des Paketes zu erfolgen.
- Für Variable vom Record Typ muß eine aktuelle Diskriminantenliste angegeben werden, falls die formalen Diskriminanten nicht mit Anfangswerten initialisiert wurden. Die Diskriminanten können nur durch eine komplette Recordzuweisung verändert werden.

- Mit der Vereinbarung von Variablen bzw. Konstanten vom Array- oder Recordtyp sind gleichzeitig Komponentenvariablen (bzw. -konstanten) vom Typ der Komponente vereinbart. Eine Komponentenvariable bzw. -konstante eines Arrays mit Komponententyp T hat einen symbolischen Namen der Form:

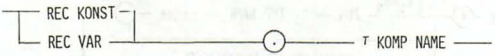
(aus ADA 33 und 48)





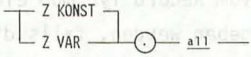
Eine Komponentenvariable bzw. -konstante eines Records hat einen symbolischen Namen der Form:

(aus ADA 33 und 48)



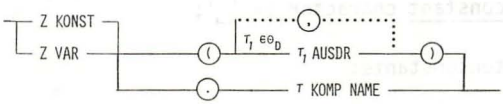
Mit der Vereinbarung von Variablen bzw. Konstanten vom Zeigertyp ist automatisch eine Variable vom referierten Typ mit dem Namen

(aus ADA 33 und 48)



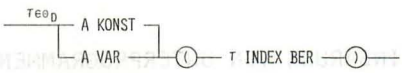
vereinbart. Man beachte, daß auch im Falle einer Zeigerkonstanten der Wert des referierten Typs verändert werden kann, also variabel ist.

- Verweist eine Zeigervariable oder -konstante auf ein Array oder Record, so sind Komponentenvariable mit dem oben angegebenen symbolischen Namen gleichzeitig mitvereinbart.



- Bei eindimensionalen Arrays, also auch Strings besteht die Möglichkeit, Teilfelder (Ausschnitte: slices) anzusprechen und damit Variable bzw. Konstante vom gleichen Array Typ mit anderen Indexgrenzen zu erhalten.

  aus ADA 33 und 48:



- Entsprechendes gilt für Zeiger auf eindimensionale Arrays.

<u>Beispiel:</u>

Variable:

```
x,y : float := 0.0;
t1,t2 : badetemperatur;
a : peripherie (drucker);
b : peripherie;        -- Diskriminante mit Platte
                       -- initialisiert
z : array (1..10,1..10) of float;
```

Mit der Variable

```
v : array (-18..307) of integer;
```

ist v(0..10) die Vektorabschnittvariable mit den Komponenten 0 bis 10.

Konstanten:

```
c : constant 1.5;
blank : constant character := '␣';
```

Komponentenkonstante:

```
m : constant array (1..2) of array (1..2) of geld
      := (((mark => 1,pfennig => 1),(2,2)),((3,3),(4,4)));
reihe : constant array (1..5) of integer := (1,2,3,4,5);
-- reihe(1) ist eine integer Konstante mit Wert 1;
-- m(1)(2) ist eine Konstante vom Typ geld, wobei die
-- Record Komponenten mark und pfennig jeweils den Wert 2
-- haben.
```

2.3 DIE VEREINBARUNG VON UNTERPROGRAMMEN

Die Vereinbarung von Unterprogrammen wurde einführend schon in 1.1 besprochen. Die vollständige Beschreibung erfolgt im Zusammenhang mit der Handhabung von Unterprogrammen in 5.1.

2.4 DIE VEREINBARUNG VON PAKETEN

Die Vereinbarung von Paketen wurde in 1.2 behandelt.

2.5 DIE VEREINBARUNG VON UNTERPROGRAMM- UND PAKETSCHEMATA

Von Unterprogrammen oder Paketen können auch sogenannte Schemata vereinbart werden, aus denen zur Übersetzungszeit entsprechende Unterprogramme oder Pakete ausgeprägt werden können (siehe 6).

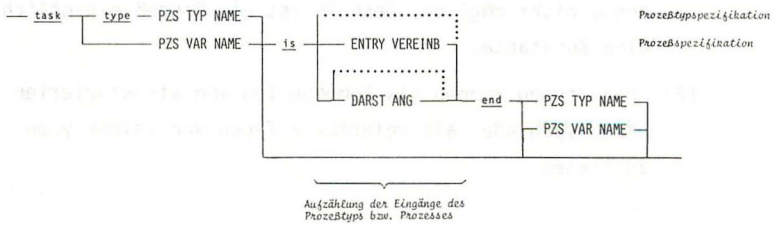
2.6 DIE VEREINBARUNG VON PROZESSEN

Prozesse sind Programmeinheiten, mit deren Hilfe die parallele Verarbeitung von Programmstücken innerhalb eines Programmes ermöglicht wird. Prozesse sind Objekte eines Prozeßtyps und werden daher wie Variable vereinbart. Der Wert einer Prozeßvariable ist also ein Prozeß vom angegebenen Prozeßtyp. Entsprechend dem Diagramm ADA 1 sind Prozesse nur als *Untereinheiten* getrennt übersetzbare Programmeinheiten (*übersetzungseinheiten*). Eine ausführliche Beschreibung der Handhabung von Prozessen findet man in Abschnitt 9.

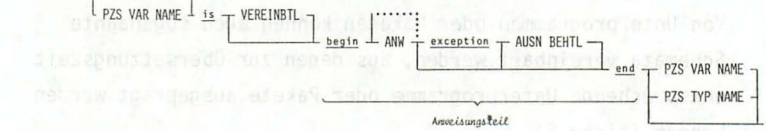
Die Vereinbarung eines Prozeßtyps besteht aus einer PROZESS-SPEZIFIKATION und einem nachfolgenden PROZESSRUMPF mit dem gleichen Prozeßtypnamen.

Syntax:

ADA 15: PROZESSSPEZIFIKATION



ADA 14: PROZESSRUMPF



Semantik:

Durch eine *Prozeßtypspezifikation* wird zunächst der Name für einen Prozeßtyp eingeführt und anschließend dessen Eingänge (siehe 9.2.1) gemäß den ENTRY VEREINBARUNGen und DARSTELLUNGSANGABEn festgelegt. Diese Eingänge bezeichnen die Schnittstellen zwischen Prozessen dieses Typs und anderen Prozessen gleichen oder anderen Typs.

Eine *Prozeßspezifikation* ist äquivalent zu einer *Prozeßtypspezifikation* mit einem anonymen Typ und anschließender Vereinbarung einer Prozeßvariablen zu diesem Typ. Bei dieser Form der Spezifikation existiert also nur ein Objekt vom angegebenen anonymen Typ.

Ein PROZESSRUMPF beschreibt die Aktionen für alle Prozesse eines Prozeßtyps. Insbesondere enthält er die für die Kommunikation mit anderen Prozessen notwendigen Schnittstellen, welche durch accept-Anweisungen festgelegt werden.

Hinweise:

(1) Operationen, Vergleiche oder Zuweisungen sind für Prozesse nicht möglich. Deshalb ist ein Prozeß eigentlich eine Konstante.

(2) Prozeßtypen können als Komponenten von strukturierten Datentypen oder als referierte Typen von Zeigertypen auftreten.

(3) Prozesse können als Eingabeargumente in Unterprogrammen auftreten. Sie können als limitierte, geschützte Typen erscheinen, deren Definition im geschützten Teil gegeben ist, und sie können anstelle von limitierten, geschützten Typen als aktuelle Schemaargumente übergeben werden.

(4) Auf Prozeßtypen sind keine Beschränkungen anwendbar.

2.7 DIE AUSNAHMEVEREINBARUNG

Die Vereinbarung von Ausnahmen wird in 7. im Zusammenhang mit der Behandlung von Ausnahmen besprochen.

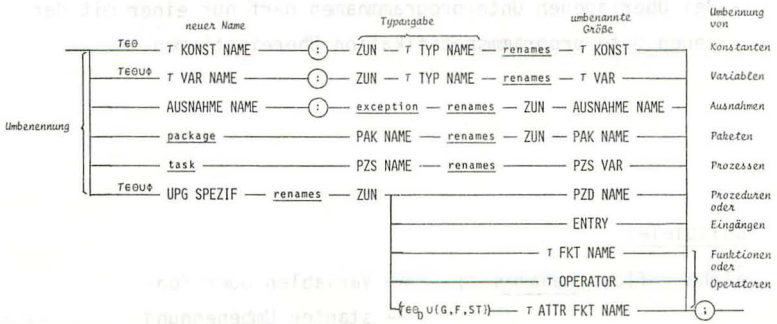
2.8 UMBENENNUNGEN

Für bekannte Größen können weitere Namen vereinbart werden.

Syntax:

Aus

ADA 18: VEREINBARUNG (VEREINB)



Semantik:

Der vor dem Wortsymbol renames (bzw. bei Konstanten, Variablen und Ausnahmen vor dem Doppelpunkt) stehende Name wird als neuer Name für die hinter renames stehende Größe eingeführt. Bei der Umbenennung von Unterprogrammen wird die gesamte Unterprogrammspezifikation verarbeitet.

Es sind folgende Punkte zu beachten:

- Bei Umbenennung von Konstanten und Variablen muß die Typangabe den gleichen Einschränkungen unterliegen wie die umbenannte Größe. Eine Komponente eines uneingeschränkten Record Typs kann nicht umbenannt werden, falls ihre Existenz von einer Diskriminante abhängt.
- Bei der Umbenennung von Unterprogrammen muß die neue formale Argumentliste die gleichen Ein- und Ausgabeargumente in der gleichen Reihenfolge wie das umbenannte Unterprogramm haben. Funktionen müssen gleichen Ergebnistyp haben. Namen der Argumente sowie eventuell vorliegende Initialisierungen können verschieden sein.
- Eine Funktion kann als Operator umbenannt werden und umgekehrt.
- Ein Eingang kann in eine Prozedur umbenannt werden.
- Bei überladenen Unterprogrammnamen darf nur einer mit der neuen Unterprogrammspezifikation übereinstimmen.

Beispiele:

```
alpha : float renames x;      -- Variablen oder Kon-
                              -- stanten Umbenennung
package car renames komplexe_arithmetik;
```

```
lm : float renames z(3,4);  -- Umbenennung eines
                            -- Matrixelementes
function "*" (x,y : complex) return complex
   renames komplexe_arithmetik.mult;
                            -- Umbenennung der Funktion
                            -- mult in das Zeichen *
```

Hinweise:

Umbenennungen dienen zur

- Abkürzung
- vorweggenommenen teilweisen Auswertung (etwa Auswahl einer Arraykomponente)
- Auflösung von Namenskonflikten: Der Name einer Größe soll für andere Zwecke gebraucht werden, die Größe selbst soll jedoch sichtbar (zugänglich) bleiben. Der alte Name wird durch eine Umbenennung nicht verborgen.

Die Umbenennung für einen Typ erfolgt durch eine Untertypvereinbarung (siehe 2.1.4).

3. AUSDRÜCKE

Ein Ausdruck ist eine Formel zur Berechnung eines Wertes, wobei im wesentlichen die übliche mathematische Formelschreibweise verwendet wird. Ein Ausdruck besteht also aus Operanden (Variable, Konstante, usw.), Operatoren (+,-,*, and, usw.) und runden Klammern ((x+5)*y). Da die Berechnung eines Ausdruckes einen Wert liefert, der von einem bestimmten Typ τ ist, müssen bei der Ausführung der einzelnen Operationen die Typen der Operanden und der jeweilige Operator zusammenpassen. So hat es in der üblichen Bedeutung sicherlich keinen Sinn, etwa den logischen Wert true und die ganze Zahl 5 zu addieren, also zu schreiben true + 5 . Trotzdem kann in ADA ein derartiger Ausdruck sinnvoll sein, dann nämlich, wenn der Operator + durch eine Operatorvereinbarung (siehe 3.9) eine zusätzliche Bedeutung bekommen hat. Man spricht dann von Überladung des Operatorzeichens, wobei sich die jeweilige Bedeutung aus den Operandentypen und dem Ergebnistyp ergibt. Wir klassifizieren in den folgenden Abschnitten die Ausdrücke gemäß ihren Ergebnistypen, da beim Auftreten eines Ausdruckes im Programm dessen Ergebnistyp das wesentliche Kennzeichen darstellt. So sprechen wir vom I AUSDRUCK, B AUSDRUCK usw., wenn der Ergebnistyp INTEGER, BOOLEAN usw. vorliegt.

Unter A u s w e r t u n g eines Ausdruckes versteht man die Bestimmung seines Wertes durch die Rechenanlage.

Für die Auswertung eines Ausdruckes gelten folgende generelle Regeln:

- Die Operanden eines Operators (ausgenommen die Kurzschlußformen and then und or else, siehe 3.1) werden ausgewertet.

  Dabei treten als Operanden auf:

- Variable (siehe 2.2)
- Konstante (ADA 36 - 42,44,46,48)
- Aggregate (ADA 43,45, 3.6, 3.7)
- geklammerte Ausdrücke
- Typkonversionen (siehe 3.10)
- qualifizierte Ausdrücke
- Funktionsaufrufe (siehe 5.4)
- Generierungen von referierten Variablen (im Falle eines Zeigerausdruckes) (siehe 3.8)

Bei einem qualifizierten Ausdruck wird durch den vorangestellten Typnamen der Typ des Ausdruckes festgelegt. Diese Qualifikation kann notwendig werden, wenn der Typ des Ausdruckes nicht eindeutig durch den Ausdruck selbst (Operanden und Operatoren) oder die Umgebung des Ausdrukkes festgelegt ist.

- Die Operatoren werden gemäß der Klammerstruktur des Ausdruckes und der für die Operatoren festgelegten Rangfolge (größere Rangnummer vor kleinerer Rangnummer) ausgeführt:

Rangnummer	Operatoren
1	and, or, xor, $\underbrace{\text{and then, or else}}_{\textit{Kurzschlußformen}}$
2	<, <=, >, >=, =, /=, in, not in
3	+, -, &
4	+, -, not (einstellig)
5	*, /, mod, rem
6	**

Dabei werden Operatoren mit gleicher Rangnummer als gleichrangig bezeichnet.

- Operatoren gleichen Ranges werden von links nach rechts

ausgeführt.

- Ein geklammerter Ausdruck wird unabhängig von nebenstehenden Operatoren ausgewertet.
- Bei einigen Operatoren wird eine Ausnahmebehandlung bei bestimmten Operandenwerten ausgelöst.

An einigen Stellen (z.B. 2.1.1.2) werden sog. s t a t i s c h e A u s d r ü c k e gefordert. Diese müssen während der Übersetzung bereits ausgewertet werden. Deshalb dürfen in statischen Ausdrücken nur folgende Operanden auftreten (definitorisch):

a) eine Literal Konstante oder ein Literal Ausdruck
b) ein Aggregat mit statischen Ausdrücken
c) eine Konstante, die durch einen statischen Ausdruck initialisiert wurde
d) die vordefinierte Funktion abs
e) ein Attribut mit statischem Wert und attributierte Funktionsaufrufe mit statischen Ausdrücken als Operanden
f) ein qualifizierter statischer Ausdruck oder eine Typkonvertierung eines statischen Ausdruckes (mit statischen Einschränkungen)
g) eine Komponente einer Record Konstante, die durch einen statischen Ausdruck initialisiert wurde
h) eine indizierte Komponente einer Feld Konstante, die durch einen statischen Ausdruck initialisiert wurde, wobei die Indexausdrücke statisch sein müssen.

Dabei sind als Operatoren zugelassen:

a) alle vordefinierten Operatoren
b) Mitgliedschaft (siehe 3.1)
c) Kurzschlußformen (siehe 3.1)

Die oben auftretende L i t e r a l K o n s t a n t e bezeichnet einen expliziten Wert eines gegebenen Typs (siehe ADA 36,37,38,39,40,42,46).

In L i t e r a l A u s d r ü c k e n sind folgende Operanden möglich:

- numerische Literal Konstanten
- Namen von numerischen Literal Konstanten
- Standardfunktionsaufruf von abs
- ein geklammerter Literal Ausdruck.

In einem Literal Ausdruck vom Typ universal_integer sind nur die vordefinierten Operatoren mit integer Operanden zulässig.

In einem Literal Ausdruck vom Typ universal_real sind folgende Operatoren zulässig:

- Vorzeichen +, -, die Funktion abs, die zweistelligen Operatoren +, -, *, / für Operanden vom Typ universal_real.
- Multiplikation von universal_real und universal_integer Operanden und umgekehrt.
- Division und Exponentiation mit universal_real Operanden als ersten und universal_integer Operanden als zweiten Operand.

Die Auswertung eines Literal Ausdruckes muß ein Ergebnis liefern, das mindestens so genau ist wie der genaueste zur Verfügung stehende numerische Typ.

Hinweise:

(1) Die Reihenfolge der Auswertung der Operanden eines Operators ist nicht festgelegt. Es ist also darauf zu achten, daß die Reihenfolge keinen Einfluß auf das Ergebnis des Operators hat. Dies könnte z.B. der Fall sein, wenn in einem Operanden ein Funktionsaufruf mit Nebenwirkung auf den anderen Operanden auftritt.

(2) Wenn die Auswertung eines statischen Ausdruckes eine Ausnahmebehandlung erfordert, so wird der betreffende statische Ausdruck durch Code ersetzt, welcher diese Ausnahmebehandlung auslöst.

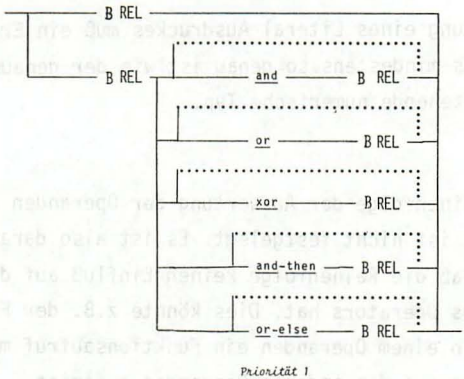
3.1 DER LOGISCHE AUSDRUCK (B AUSDRUCK)

Logische Aussagen wie Vergleiche, die logischen Konstanten true und false ("wahr" und "falsch") können z.B. durch and und or (logisches "und" und "oder") zu neuen logischen Aussagen verknüpft werden. Solche Gebilde werden hier logische Ausdrücke genannt. Dabei beschränken wir uns in diesem Abschnitt auf die Standardbedeutung aller auftretenden Operatoren. (Andernfalls siehe 3.9).

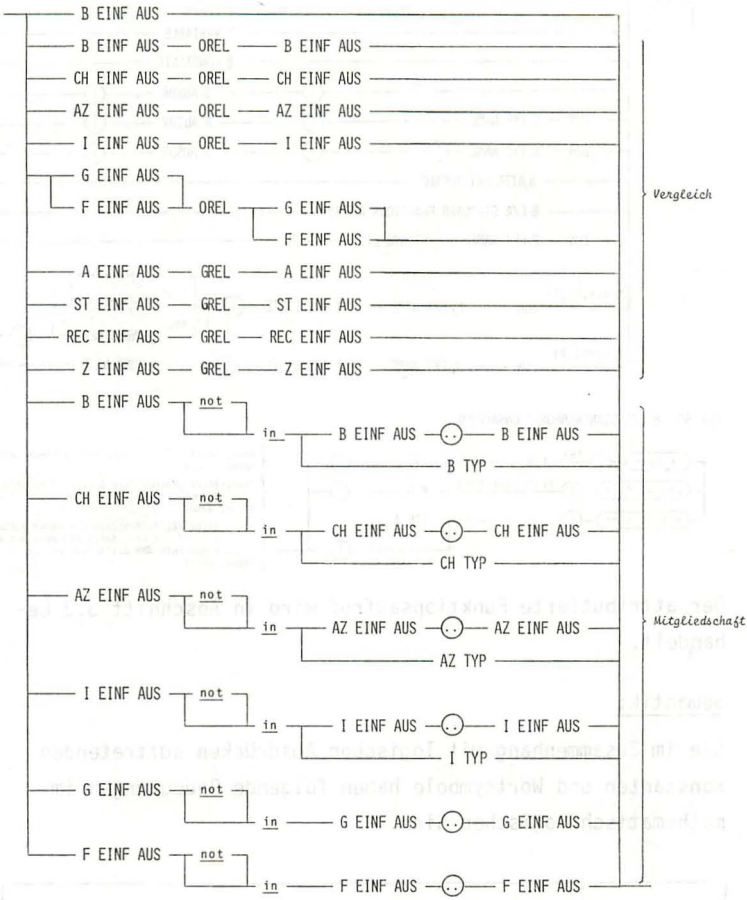
Die Auswertung eines logischen Ausdruckes bewirkt die Feststellung, ob die angegebene mathematisch logische Aussage den Wert true oder false hat.

Syntax:

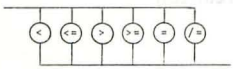
ADA 28: B AUSDRUCK (B AUSDR, B A)



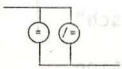
ADA 29: B RELATION (B REL)



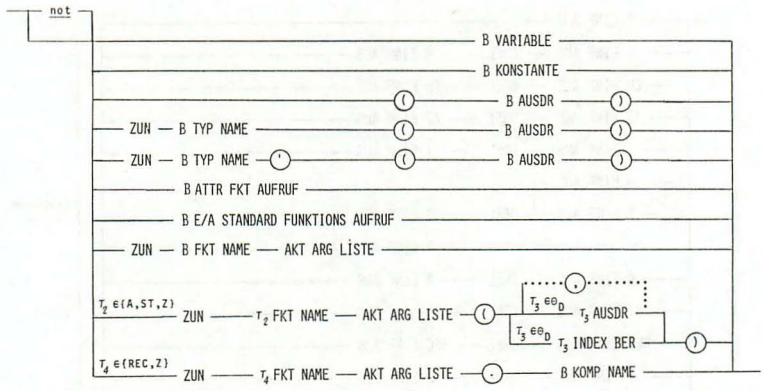
ORDNUNGSRELATION (OREL)



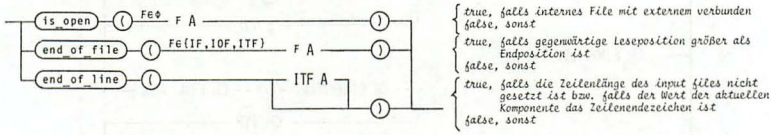
GLEICHHEITSRELATION (GREL)



ADA 30: B EINFACHER AUSDRUCK (B EINF AUS)



ADA 55: B E/A STANDARDFUNKTIONSAUFRUF



Der attributierte Funktionsaufruf wird in Abschnitt 3.3 behandelt.

Semantik:

Die im Zusammenhang mit logischen Ausdrücken auftretenden Konstanten und Wortsymbole haben folgende Bedeutungen im mathematisch logischen Sinn:

Symbol	Bedeutung
true	"wahr" } logische Konstanten
false	"falsch" } logische Konstanten
not	Negation
and	Konjunktion

Symbol	Bedeutung
or	einschließliche Disjunktion
xor	ausschließliche Disjunktion
and then or else	sog. Kurzschlußformen

Die logischen Operatoren sind durch folgende Wertetabellen definiert:

α	β	not α	α and β	α or β
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

α	β	α xor β	α and then β	α or else β
true	true	false	true	true
true	false	true	false	true
false	true	true	false	true
false	false	false	false	false

der Operand β wird nicht ausgewertet

der Operand β wird nicht ausgewertet

Standardfunktionen

Die Standardfunktionen (ADA 55) haben folgende Bedeutung:
(α ist Wert für F AUSDRUCK, $F \in \Phi$, Φ={IF,OF,IOF,ITF,OTF}
β ist Wert für T AUSDRUCK, $T \in$ {IF,IOF,ITF}
γ ist Wert für ITF AUSDRUCK)

Standard-funktion	logischer Wert	Bedeutungen und Bedingungen
is_open(α) bzw. is_open(file=>α)	true	das interne File α ist mit einem externen File verbunden
	false	sonst
end_of_file(β) bzw. end_of_file(file=>β)	true	die aktuelle Leseposition von β ist größer als die Endposition
	false	sonst
end_of_line(γ) bzw. end_of_line(file=>γ)	true	die Zeilenlänge des input-Files ist nicht gesetzt bzw. der Wert der aktuellen Komponente ist das Zeilenendezeichen
	false	sonst

Ein Vergleich (ADA 29: B RELATION) ist eine logische Aussage, die den Wert true hat, wenn der Operator die Anordnung der Operanden (die vom selben Typ sein müssen) in der durch ihren Typ bestimmten Ordnungsbeziehung beschreibt. Andernfalls hat der Vergleich den Wert false.

Die Vergleichsoperatoren (ORDNUNGSRELATION) haben folgende Gültigkeit und Bedeutung:

Typ	Vergleichsoperatoren <	<=	>	>=	=	/=	Bemerkungen
B	<	≦	>	≧	=	≠	false < true
CH	<	≦	>	≧	=	≠	Ordnung gemäß ASCII-Zeichensatz
AZ	<	≦	>	≧	=	≠	Ordnung gemäß Reihenfolge in der Typdefinition
I	<	≦	>	≧	=	≠	natürliche Ordnung der ganzen Zahlen
G od. F	<	≦	>	≧	=	≠	

Das Modellintervall A eines reellen Wertes a ist das kleinste Intervall mit Modellzahlen als Grenzen, das den reellen Wert enthält.

Entsprechend dem Typ werden beiden Operanden a, b ihre Modellintervalle A, B zugeordnet und folgende Fälle unterschieden:

(a) $A \cap B = \emptyset$ => Ergebnis entsprechend math. Definition

(b) A = a, B = b, A = B => Ergebnis entsprechend math. Definition

(c) $A \cap B$ = x (Modellzahl) => Ergebnis entsprechend math. Definition durch Vergleich der geg. Operanden a,b oder durch Vergleich des ersten Operanden mit sich selbst.

(d) $A \cap B$ enthält mehr als einen reellen Wert => Ergebnis maschinenabhängig

(e) Die Bildung von A oder B ergibt Überlauf => Ergebnis maschinenabhängig

Typ	Vergleichsoperatoren <	<=	>	>=	=	/=	Bemerkungen
In diesem Fall kann, muß aber nicht, die Ausnahmebehandlung numeric_error erfolgen.							
A					=	≠	für nichtleere Felder gleichen Typs komponentenweise erklärt, wobei die beiden Felder zueinander passen müssen, d.h. beginnend mit jeweils der unteren Grenze der Indexbereiche lassen sich aufeinanderfolgende Komponenten paarweise zuordnen. Leere Felder vom selben Typ sind immer gleich.
A eindimensional	<	≦	>	≧	=	≠	für diskreten Komponententyp erklärt, dabei wird lexikographische Ordnung geprüft. Ein leerer Vektor ist kleinster Vektor.
ST	<	≦	>	≧	=	≠	
REC					=	≠	für nichtleere Records gleichen Typs komponentenweise erklärt, wobei die beiden Records zueinander passen müssen, d.h. beginnend bei der 1. Komponente paarweise gleiche Komponentennamen haben. Leere Records vom selben Typ sind immer gleich
Z					=	≠	zwei Zeigerwerte sind gleich, wenn sie entweder beide auf dasselbe Objekt zeigen oder beide den Wert null haben.

Typ	Vergleichsoperatoren <	<=	>	>=	=	/=	Bemerkungen
geschützte Typen und strukturierte Typen mit geschütztem Typ als Komponententyp	X	X	X	X	X	X	nicht vergleichbar

Typ	Mitgliedschaft in	not in
B CH AZ I G F	true, falls der Wert des linken Operanden innerhalb des rechtsstehenden Bereichs liegt. Der Wert muß vom gleichen Typ sein wie die Grenzen des Bereichs bzw. wie der Grundtyp des Untertyps.	false, ...
	false, sonst	true, sonst

Beispiele:

1. Seien a,b,c,d logische Variable mit den Werten a:= true, b:= true, c:= false, d:= false, so ergibt die Auswertung des logischen Ausdruckes

$$\underbrace{\underbrace{(a\ \underline{xor}\ b)}_{false}\ \underline{or}\ (\underline{not}\ a\ \underline{and}\ \underbrace{(c\ \underline{or}\ \underline{not}\ d)}_{true})}_{false}$$

den Wert false.

2. Es seien x,y Variable vom Typ float_real. Dann gibt der logische Ausdruck (Vergleich von zwei logischen Aussagen)

   (y >= -x/8.5 + 1.5) = (y >= x/2.7)

   an, ob der durch das Paar (x,y) in der Zahlenebene beschriebene Punkt gleichzeitig oberhalb der beiden Geraden y = -x/8.5 + 1.5, y = x/2.7 oder echt unterhalb dieser Geraden liegt.

3. Sei p eine Zeigervariable auf einen Record Typ mit einer Komponente "daten". So erlaubt der Ausdruck

   p /= <u>null</u> <u>and</u> <u>then</u> p.daten /= x

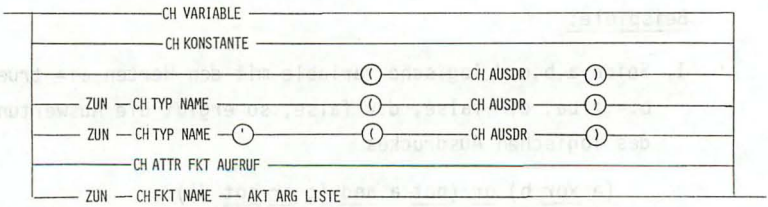
   das Durchsuchen einer entsprechenden Liste nach dem Wert x.

3.2 DER ZEICHEN AUSDRUCK (CH AUSDRUCK)

Zeichen sind Werte vom Typ Character. Dieser Ausdruck besitzt keine Standardoperatoren und keine Standardfunktionen.

<u>Syntax:</u>

ADA 28: CH AUSDRUCK (CH AUSDR, CH A)



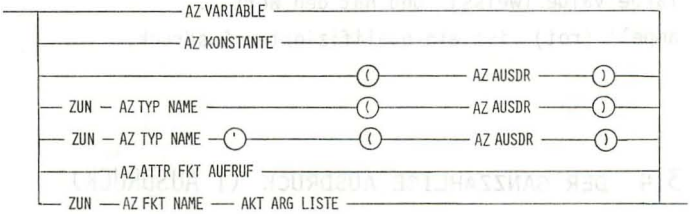
Der attributierte Funktionsaufruf wird in Abschnitt 3.3 behandelt.

3.3 DER AUSDRUCK VOM AUFZÄHLUNGSTYP (AZ AUSDRUCK)

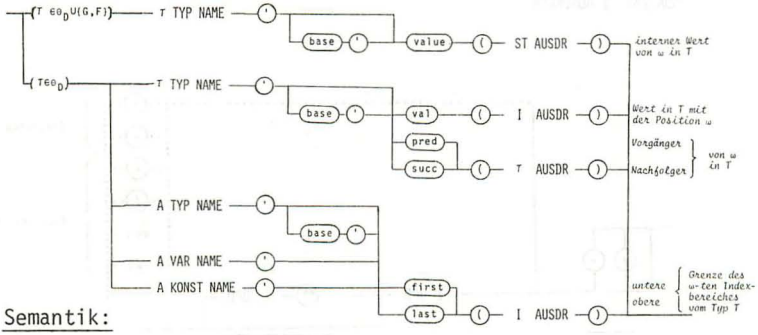
Wie der Zeichen Ausdruck besitzt auch der Ausdruck vom Aufzählungstyp weder Standardoperatoren noch Standardfunktionen.

Syntax:

ADA 28: AZ AUSDRUCK



ADA 35: T ATTRIBUTIERTER FUNKTIONSAUFRUF (T ATTR FKT AUFRUF) $T \in \Theta_D \cup \{G,F,ST\}$
$\Theta_D = \{B,CH,AZ\}$



Semantik:

Attributierte Funktionsaufrufe sind Aufrufe von typabhängigen Standardfunktionen, die jeweils mit dem entsprechenden Typnamen qualifiziert werden.

Tritt das Attribut base auf, so wirkt der Aufruf auf den zugrundeliegenden Basistyp.

Beispiele:

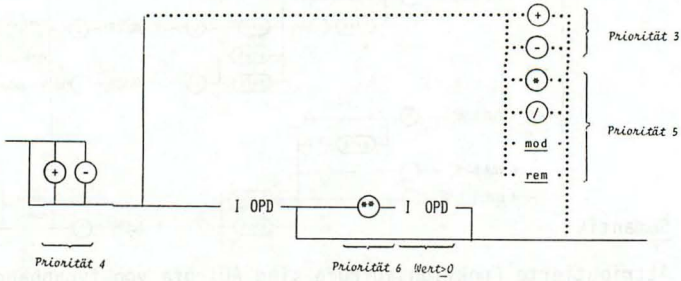
```
type ampel is (rot,rotgelb,gruen,gelb);
type farbe is (blau,rot,gelb,weiss,grau,schwarz);
subtype grauton is farbe range weiss .. schwarz;
```

ampel'succ (rotgelb) hat den Wert gruen.
grauton'value (weiss) hat den Wert Ø.
grauton'base'value (weiss) stimmt überein mit
farbe'value (weiss) und hat den Wert 3.
ampel' (rot) ist ein qualifizierter Ausdruck.

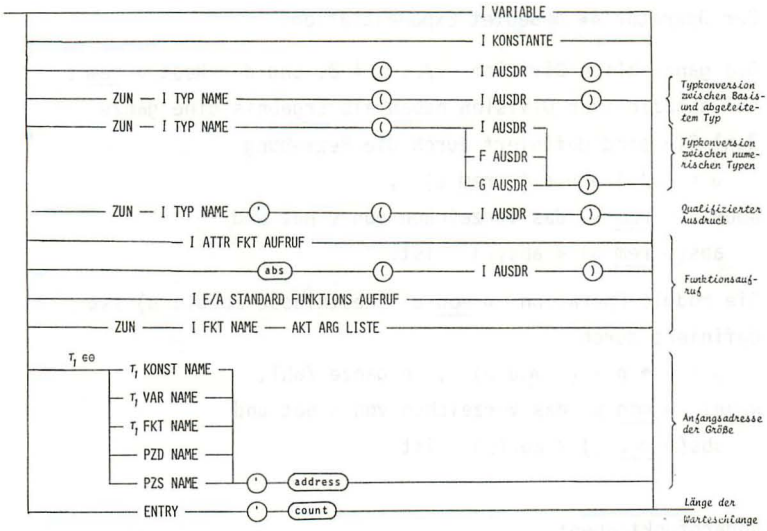
3.4 DER GANZZAHLIGE AUSDRUCK (I AUSDRUCK)

Der I AUSDRUCK hat folgenden syntaktischen Aufbau:

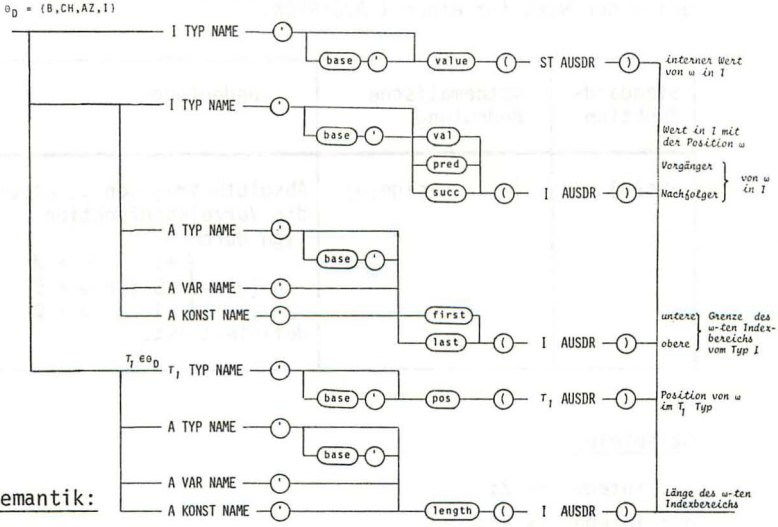
ADA 28: I AUSDRUCK



ADA 31: I OPERAND (I OPD)
Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z}



ADA 35: I ATTRIBUTIERTER FUNKTIONSAUFRUF (I ATTR FKT AUFRUF)
Θ_D = {B,CH,AZ,I}



Semantik:

Seien ν,μ Operanden vom Typ integer. Dann gilt:

- Die Operatoren +, -,* haben die übliche mathematische Bedeutung.

- Der Operator ** bedeutet Exponentiation.
- Die ganzzahlige Division ν/μ, $\mu \neq \emptyset$, und der Rest ν <u>rem</u> μ bei ganzzahliger Division haben als Ergebnis eine ganze Zahl. Sie sind definiert durch die Beziehung

  $\nu = (\nu/\mu) * \mu + (\nu \ \underline{\text{rem}} \ \mu)$,

  wobei ν <u>rem</u> μ das Vorzeichen von ν hat und

  $\text{abs}(\nu \ \underline{\text{rem}} \ \mu) < \text{abs}(\mu)$ ist.
- Die modulo-Operation ν <u>mod</u> μ (Restklasse modulo μ) ist definiert durch

  $\nu = \mu * n + (\nu \ \underline{\text{mod}} \ \mu)$, n ganze Zahl,

  wobei ν <u>mod</u> μ das Vorzeichen von μ hat und

  $\text{abs}(\nu \ \underline{\text{mod}} \ \mu) < \text{abs}(\mu)$ ist.

<u>Standardfunktionen:</u>

Sei ν der Wert für einen I AUSDRUCK.

Standard-funktion	mathematische Bedeutung	Bedeutung
abs(ν)	$\lvert\nu\rvert := \nu \cdot \text{sign}(\nu)$	Absolutbetrag von ν, wobei die Vorzeichenfunktion sign durch $\text{sign}(\nu) := \begin{cases} +1 & \nu > \emptyset \\ 0 \text{ für} & \nu = \emptyset \\ -1 & \nu < \emptyset \end{cases}$ definiert ist.

<u>Beispiele:</u>

```
i : integer := 2;
j : integer := -3;
k : integer := 5;
l,m,n : integer;
l:= i * j;                l erhält den Wert -6 zugewiesen
```

m:= k/j;	m erhält den Wert -1 zugewiesen
n:= k <u>mod</u> j;	n erhält den Wert -1 zugewiesen
m:= k <u>rem</u> j;	m erhält den Wert +2 zugewiesen.

Hinweise:

(1) Bei den Operationen +, -, *, / und ** erfolgt die Ausnahmebehandlung numeric_error, wenn das Ergebnis außerhalb des Ergebnistyps liegt, insbesondere bei ν/μ, ν <u>rem</u> μ, ν <u>mod</u> μ, falls $\mu = \emptyset$ ist.

(2) Für die ganzzahlige Division und die Operatoren <u>rem</u> und <u>mod</u> gelten folgende Eigenschaften:

$(-\nu/\mu) = -(\nu/\mu) = (\nu/-\mu)$

ν <u>rem</u> $(-\mu) = \nu$ <u>rem</u> μ

$(-\nu)$ <u>rem</u> $\mu = -(\nu$ <u>rem</u> $\mu)$

ν <u>mod</u> $\mu = (\nu + k * \mu)$ <u>mod</u> μ für alle ganzen Zahlen k

ν rem $\mu = \nu$ mod μ für sign(ν) = sign(μ) .

(3) Bei negativem Exponenten bei der Exponentiation erfolgt die Ausnahmebehandlung constraint_error.

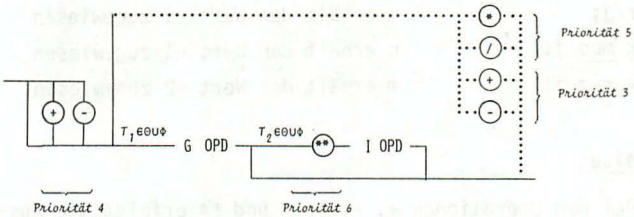
3.5 DER REELLE AUSDRUCK

Entsprechend den beiden reellen Typen, dem Gleitpunkt- und dem Festpunkttyp, gibt es auch zugehörige Ausdrücke.

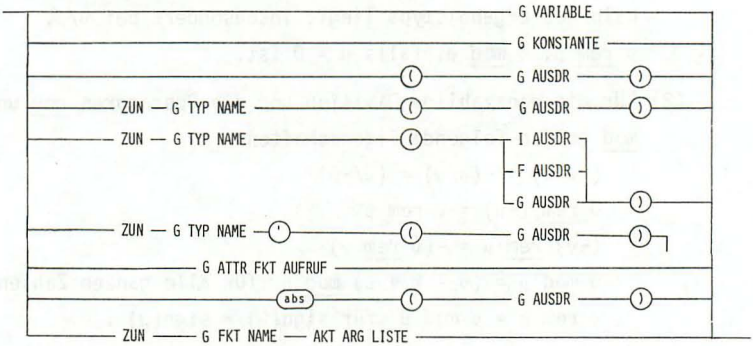
3.5.1 DER GLEITPUNKTAUSDRUCK (G AUSDRUCK)

Der G AUSDRUCK hat folgenden syntaktischen Aufbau:

ADA 28: G AUSDRUCK



ADA 31: G OPERAND (G OPD)



Semantik:

- Die Gleitpunktoperatoren +, -, *, / haben die übliche mathematische Bedeutung, abgesehen davon, daß das Ergebnis im allgemeinen nicht Element des verwendeten Gleitpunktsystems T sein wird. In diesem Falle gilt folgendes:

  Seien a,b ∈ T vom gewählten Gleitpunkttyp und □a,□b ∈ M die entsprechenden Modellzahlen (wegen möglichem Basiswechsel muß evtl. konvertiert werden).

  ◇a bzw. ◇b sei das kleinste a bzw. b enthaltende Modellintervall (zugeordnetes Modellintervall) (Nicht definiert, falls |a| > T'LARGE).

  Es gilt:

  $\bigwedge_{a \in M} \Diamond a := [a,a]$, d.h. das Modellintervall einer Modellzahl ist ein Punktintervall

  $\bigwedge_{[a_1,a_2]=A \in \mathbb{I}(T)} \Diamond A := [\nabla a_1, \Delta a_2]$

∇ bzw. Δ bezeichne die monotone nach unten bzw. oben gerichtete Rundung [35].

Zwei Elemente $a,b \in T$ werden derart miteinander verknüpft, daß gilt:

$$\text{(RG')} \qquad \bigwedge_{\circ \in \{+,-,*,/\}} \bigwedge_{a,b \in T} a \boxdot b \in \Diamond(\Diamond a \diamond \Diamond b) \quad ,$$

wobei mit $\diamond$ die der Verknüpfung $\circ$ entsprechende Intervallverknüpfung bezeichnet sei.

Intervalle schreiben wir in der Form
$[a,b] := \{x \in \mathbb{R} \mid a \leq x \leq b\}$.

Die Berechnung der Schranken der Gleitpunktverknüpfungen erfolgt also in 3 Schritten:

1. Jedem Operanden wird ein Modellintervall zugeordnet.
2. Diese Modellintervalle werden intervallarithmetisch verknüpft.
3. Dem in Schritt 2. entstehenden Intervall wird ein Modellintervall vom Ergebnistyp der zugrundeliegenden Operation zugeordnet.

- Die Exponentiation a**n ist definiert als a^n durch:

$$a{**}n := \begin{cases} \underbrace{a*a* \ldots *a}_{n\text{-mal}} & n > 0 \\ 1 & \text{für} \quad n = 0 \\ 1/\underbrace{(a*a* \ldots *a)}_{-n\text{-mal}} & n < 0 \end{cases}$$

<u>Beispiel:</u>

Der Gleitpunktausdruck

((12192.Ø * x - 32257.Ø) * x - 85344.Ø) * x + 225799.Ø

hat für x = 2.645752 den Wert 3.Ø5639Ø5536e-8 .

Hinweise:

(1) Bei den Operationen +, -, *, / und ** erfolgt die Ausnahmebehandlung numeric_error wenn das Ergebnis außerhalb des Ergebnistyps liegt.

(2) Bei der Ausführung der Operationen kann bei Intervallüberlauf die Ausnahmebehandlung numeric_error erfolgen (muß aber nicht).

(3) Für zwei Modellzahlen a,b ∈ M wird (RG') zu

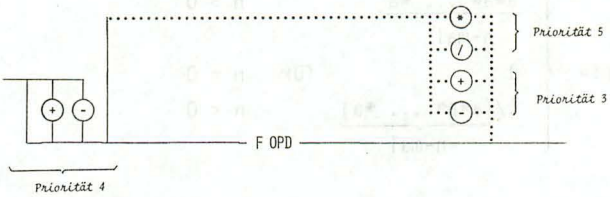
$$\bigwedge_{\circ \in \{+,-,*,/\}} \bigwedge_{a,b \in M} a \boxdot b \in \Diamond(a \circ b) \quad ,$$

d.h. a ⊡ b liegt in einem Intervall, dessen Grenzen zwei benachbarte Modellzahlen sind.

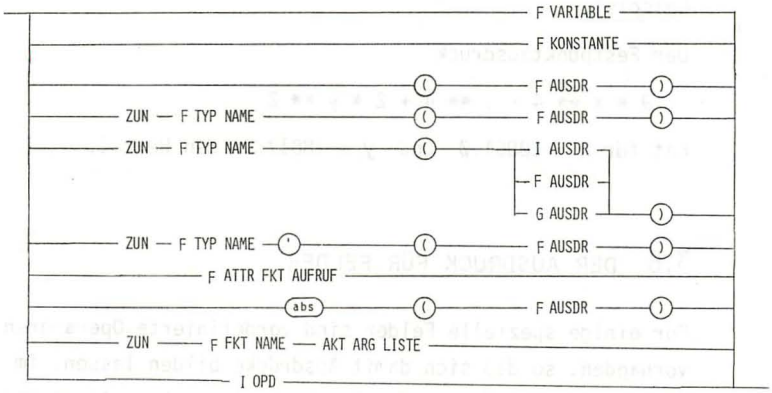
3.5.2 DER FESTPUNKTAUSDRUCK (F AUSDRUCK)

Der F AUSDRUCK hat folgenden syntaktischen Aufbau:

ADA 28: F AUSDRUCK



ADA 31: F OPERAND (F OPD)



Semantik:

Die Festpunktoperatoren +, -, * und / haben die übliche mathematische Bedeutung, wobei bez. Genauigkeit das für Gleitpunktoperationen Gesagte gilt. Die zulässigen Paare von I und F OPERANDen seien für * und / tabellarisch dargestellt:

Operator	Operandentyp links	rechts	Ergebnistyp	Bemerkungen
*	F	I	wie links	entspricht wiederholter Addition und ist deshalb exakt.
	I	F	wie rechts	
	F	F	universeller Festpunkttyp	exakt, muß explizit in geeigneten Typ konvertiert werden.
/	F	I	wie links	i.a. nicht exakt
	F	F	universeller Festpunkttyp	das Ergebnis muß explizit in geeigneten Typ konvertiert werden.

Dabei können die auftretenden F OPERANDEN durchaus verschiedene Festpunkttypen sein.

Beispiel:

Der Festpunktausdruck

```
9 * x ** 4 - y ** 4 + 2 * y ** 2
```

hat für x = 1Ø864.Ø und y = 18817.Ø den Wert 1.

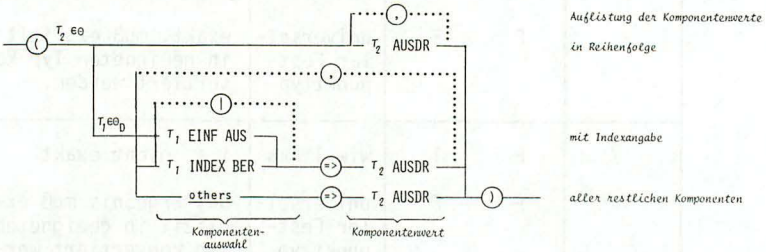
3.6 DER AUSDRUCK FÜR FELDER

Für einige spezielle Felder sind vordefinierte Operatoren vorhanden, so daß sich damit Ausdrücke bilden lassen. Im allgemeinen muß jedoch der Benutzer eigene Operatoren vereinbaren, etwa dann, wenn er Matrizen numerisch verknüpfen will (siehe 3.9).

Der allgemeine A AUSDRUCK besteht demnach lediglich aus einer Aufzählung von Operanden (siehe ADA 31). Dabei tritt als Operand das sogenannte A AGGREGAT auf. Damit kann man den Wert eines Feldes angeben:

Syntax:

ADA 43: A AGGREGAT
Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Θ_D = {B,CH,AZ,I}



Semantik:

Ein Array Aggregat stellt einen Wert eines Feldes dar, indem alle Komponenten explizit (in Form von Ausdrücken) angegeben werden. Dazu gibt es mehrere Möglichkeiten:

(a) Auflistung mit Indexangabe

Ein Aggregat mit einer einzigen Komponente muß in dieser Form angegeben werden.

Ist die *Komponentenauswahl* ein einfacher Ausdruck, so ist dessen Wert der angesprochene Indexwert.

Ist die Auswahl ein Indexbereich, so sind alle darin enthaltenen Indexwerte angesprochen.

Allen durch eine *Komponentenauswahl* gekennzeichneten Feldkomponenten wird der Wert des nach => stehenden Ausdruckes zugewiesen. Die Reihenfolge der Auflistung ist beliebig.

Die Indexgrenzen eines solchen Aggregates sind durch den kleinsten und größten auftretenden Indexwert gegeben.

Der Wert jeder *Komponentenauswahl* muß statisch bestimmbar sein, wenn das Aggregat mehr als einen Komponentenwert enthält.

Alle durch *Komponentenauswahl* auftretenden Indexwerte müssen verschieden sein.

Beispiel:

Das Aggregat

```
(1 .. 1Ø => 1.Ø,11 .. 2Ø => Ø.Ø)
```

stellt den Wert eines Vektors der Länge 2Ø dar.

(b) Auflistung mit others

Allen Komponenten wird der Wert des nach => stehenden Ausdruckes zugewiesen. Der Indexbereich ist durch den Kontext bestimmt.

Derartige Aggregate dürfen nur in folgenden Zusammenhängen auftreten:

1. Das Aggregat ist aktuelles Argument zu einem formalen Argument mit beschränktem Array Untertyp.
2. Das Aggregat erscheint in einer return-Anweisung als der Ausdruck, der den Ergebniswert einer Funktion

angibt, deren Ergebnistyp ein A Untertyp mit beschränktem Indexbereich ist.

3. Das Aggregat ist entweder durch einen beschränkten Feldtyp qualifiziert oder wird bei der Generierung einer referierten Variablen von einem A Untertyp mit beschränktem Indexbereich verwendet.
4. Das Aggregat tritt selbst innerhalb eines Aggregats auf, welches wiederum in 1., 2. oder 3. auftritt.

In allen 4 Fällen sind die Indexgrenzen des Aggregats durch die im Kontext verfügbaren Indexangaben bestimmt.

Beispiel:

Das Aggregat

(others => Ø.Ø)

weist allen Feldkomponenten den Wert Ø zu.

(c) Auflistung in Reihenfolge

Die Indexgrenzen des Aggregats sind wie in den in (b) beschriebenen Fällen 1. bis 4. ebenfalls durch die im Kontext verfügbaren Indexangaben bestimmt. Andernfalls ist die untere Grenze durch s'first bestimmt, wobei s der Index Untertyp ist. Die obere Grenze ist durch die Anzahl der Komponenten bestimmt. Diesen Feldelementen werden der Reihe nach die Werte der aufgelisteten Ausdrücke zugewiesen.

Beispiel:

Das Aggregat

(Ø.1,Ø.5,1.2)

gibt den Wert eines Vektors mit 3 Komponenten an.

(d) Gemischte Auflistung

In einem Aggregat darf die Auflistung mit Indexangabe und in Reihenfolge nicht gleichzeitig erfolgen. Die Auflistung mit others bezieht sich im Zusammenhang mit (a)

bzw. (c) auf die restlichen Komponenten.

Beispiel:

Das Aggregat

```
(1 .. 1Ø => 1.Ø, others => Ø.Ø)
```

gibt den Wert eines Vektors an, bei dem 1Ø Komponenten den Wert 1, alle anderen (vorstehenden bzw. nachfolgenden) den Wert Ø haben.

(e) Ein Aggregat eines n-dimensionalen Feldes wird als eindimensionales Aggregat geschrieben, dessen Komponenten selbst wieder (n-1)-dimensionale Feldwerte sind.

Beispiel:

Das Aggregat

```
(1 .. 1Ø => (1 .. 1Ø => Ø.Ø))
```

stellt den Wert einer 1Ø×1Ø-Matrix dar, deren Komponenten alle den Wert Ø haben.

Hinweise:

1. Die Reihenfolge der Auswertung der den einzelnen Komponenten zugeordneten Ausdrücken liegt nicht fest. Programme, die eine solche Reihenfolge voraussetzen, sind deshalb fehlerhaft.

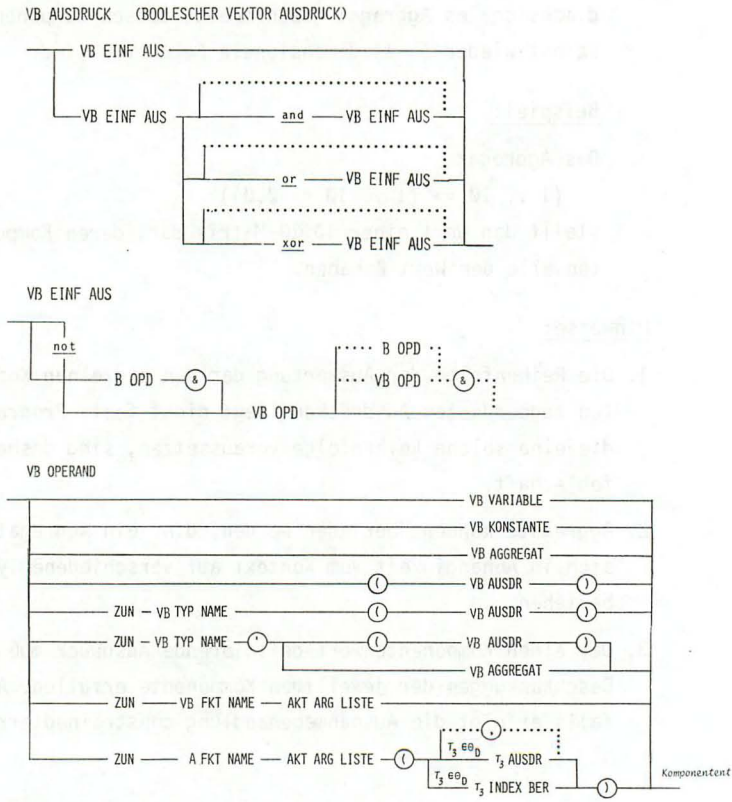
2. Aggregate können überladen werden, d.h. ein Aggregat kann sich in Abhängigkeit vom Kontext auf verschiedene Typen beziehen.

3. Der einen Komponentenwert definierende Ausdruck muß die Beschränkungen der jeweiligen Komponente erfüllen. Andernfalls erfolgt die Ausnahmebehandlung constrained_error.

3.6.1 DER AUSDRUCK FÜR FELDER MIT KOMPONENTEN-TYP boolean

Wir bezeichnen im folgenden mit VB (VEKTOR BOOLEAN) einen eindimensionalen A Typ mit Komponententyp boolean. Die entsprechenden Syntaxvariablen sind also aus dem allgemeinen A Typ ableitbar.

Syntax:



Semantik:

Die logischen Operatoren not, and, or, xor werden komponenten-

weise ausgeführt. Dazu müssen die Operanden vom gleichen (nicht leeren) eindimensionalen A Typ mit Komponententyp boolean und gleicher Komponentenanzahl sein.
Das Ergebnis ist vom selben VB Typ wie der (die) VB Operand (VB Operanden).

Die Operation & (Konkatenation, Aneinanderhängen) fügt die beiden Operanden zu einem einzigen Vektor zusammen, in dem der linke Operand den vorderen und der rechte Operand den hinteren Teil bildet:

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \& \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} := \begin{pmatrix} a_1 \\ \vdots \\ a_n \\ b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Für einen eindimensionalen A Typ T mit dem Komponententyp C und dem Indexbereich

index <u>range</u> <>

wird die Wirkung der Operation & durch folgende drei Funktionen beschrieben:

```
low : constant index := index'first;

function "&" (x,y : T) return T is
   result : T(low .. index'val(index'pos(low) + (x'length +
            y'length - 1)));
begin
   result(low .. index'val(index'pos(low)+(x'length-1))):=x;
   result(index'val(index'pos(low)+x'length) .. result'last)
           := y;
   return result;
end;

function "&" (x : C; y : T) return T is
begin
   return (low => x) & y;
end;
```

```
function "&" (x : T; y : C) return T is
begin
   return x & (low => y);
end;
```

Hinweise:

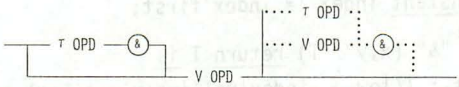
(1) Die Operanden der Operatoren and, or, xor brauchen nicht dieselben Indexgrenzen zu besitzen, aber sie müssen die gleiche Anzahl von Komponenten haben. Andernfalls erfolgt die Ausnahmebehandlung constraint_error.

3.6.2 DER AUSDRUCK FÜR EINDIMENSIONALE FELDER

Wir bezeichnen im folgenden mit V (VEKTOR) einen eindimensionalen A Typ mit Komponententyp *T*.

Syntax:

V AUSDRUCK (VEKTOR AUSDRUCK)



V OPERAND

analog VB OPERAND in 3.6.1

Semantik:

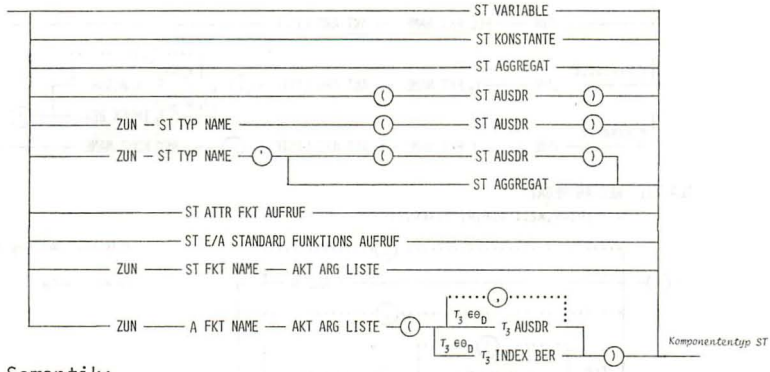
Analog 3.6.1.

3.6.3 DER AUSDRUCK FÜR ZEICHENKETTEN

Syntax:

ADA 28: ST AUSDRUCK

ADA 31: ST OPERAND



Semantik:

Die Konkatenation & hat die Bedeutung, wie in 3.6.1 beschrieben.

Beispiel:

"wer dauernd telefoniert"&",bringt nichts zu papier."

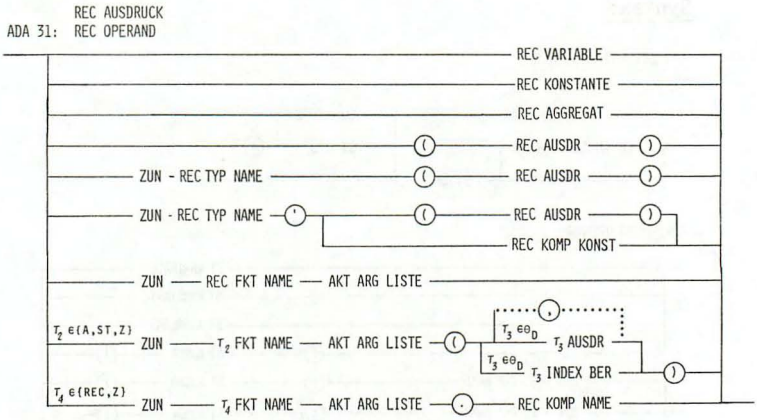
('a','d','a') ist ein Aggregat mit dem Wert "ada" .

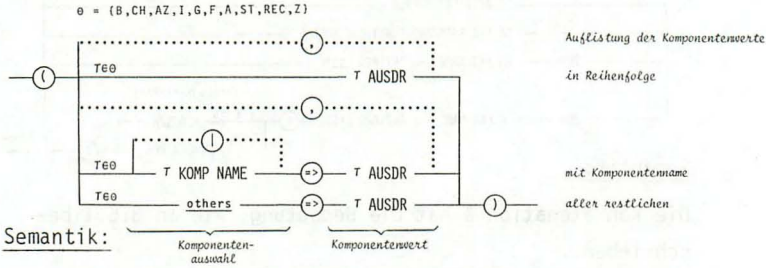
3.7 DER RECORD AUSDRUCK

Für Records sind standardmäßig keine Operatoren vorgesehen.

Der Record Ausdruck hat deshalb einen sehr einfachen Aufbau:

Syntax:





Semantik:

Ein Record Aggregat stellt einen Wert eines Records dar, indem für jede Komponente (einschließlich Diskriminanten) ein expliziter Wert angegeben wird, unabhängig davon, ob eine Komponente bereits einen Wert besitzt oder nicht. Der Wert für eine Diskriminante muß durch einen statischen Ausdruck angegeben werden. Für die Angabe der Komponentenwerte gibt es folgende Möglichkeiten:

(a) Auflistung mit Komponentenname
Allen aufgelisteten, durch | getrennten Komponentennamen wird der Wert des nach => stehenden Ausdruckes zugewiesen. Deshalb müssen alle diese Komponenten von demsel-

ben Typ sein. Die Reihenfolge der aufgelisteten Komponentennamen ist beliebig.

Beispiel:

```
type complex is
   record realteil : real := 1.Ø;
          imaginaerteil : real := Ø.Ø;
   end record;
```

Das Aggregat

```
(realteil | imaginaerteil => Ø.Ø)
```

stellt den Wert eines Records mit 2 Komponenten dar, die beide den Wert Ø erhalten.

(b) Auflistung mit others

Wenn alle Komponenten denselben Typ haben und denselben Wert erhalten sollen, kann dies durch Auflistung mit others erreicht werden.

Beispiel:

Das Aggregat

```
(others => Ø.Ø)
```

hat im Zusammenhang des Typs complex dieselbe Wirkung wie das Beispiel in (a).

(c) Auflistung in Reihenfolge

Den Recordkomponenten werden in der durch die Recordvereinbarung festgelegten Reihenfolge der Reihe nach die Werte der aufgelisteten Ausdrücke zugewiesen.

Beispiel:

```
type datum is
   record geburtstag,todestag : integer range 1 .. 31;
          geburtsmonat,sterbemonat : monatsname;
          geburtsjahr,sterbejahr : integer range
                                   -5ØØØ .. 3ØØØ;
```

end record;

Das Aggregat

(12,7,oktober,oktober,1882,1942)

stellt einen Wert des Recordtyps datum dar.

(d) Gemischte Auflistung

Dabei muß die Auflistung in Reihenfolge zuerst erfolgen. Die Auflistung mit others bezieht sich im Zusammenhang mit (a) und (c) auf die restlichen noch verbleibenden Komponenten.

Beispiel:

Das Aggregat

(12,7,geburtsjahr => 1882,sterbejahr => 1942,
others => oktober)

hat dieselbe Wirkung wie das Beispiel in (c).

Hinweise:

(1) Die Reihenfolge der Auswertung der den einzelnen Komponenten zugeordneten Ausdrücken liegt nicht fest. Programme, die eine solche Reihenfolge voraussetzen, sind deshalb fehlerhaft.

(2) Aggregate können überladen werden, d.h. ein Aggregat kann sich in Abhängigkeit vom Kontext auf verschiedene Typen beziehen.

(3) Der einen Komponentenwert definierende Ausdruck muß die Beschränkungen der jeweiligen Komponente erfüllen. Andernfalls erfolgt die Ausnahmebehandlung constraint_error.

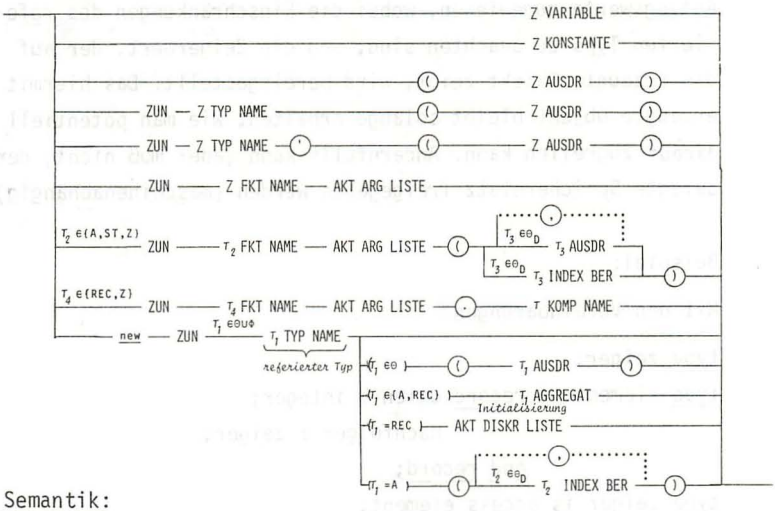
(4) Bei Auflistung in Reihenfolge müssen die Werte für Diskriminanten zuerst angegeben werden, da der Diskriminantenteil zuerst angegeben wird. Zudem müssen die Diskriminantenwerte in derselben Reihenfolge wie im Diskriminantenteil angegeben werden.

3.8 DER AUSDRUCK FÜR ZEIGER

Für Zeiger sind standardmäßig keine Operatoren vorgesehen. Der Zeiger Ausdruck hat deshalb einen sehr einfachen Aufbau:

Syntax:

ADA 31: Z AUSDRUCK
Z OPERAND



Semantik:

Die Ausführung der Speicherbelegungsfunktion new erzeugt ein Objekt vom referierten Typ und liefert als Wert einen Zeiger auf dieses Objekt. Vor der Objekterzeugung wird zunächst jeder Ausdruck, jedes Aggregat, jede Diskriminantenfestlegung, jede Indexbereichseinschränkung ausgewertet. Dann wird ein neues Objekt vom referierten Typ erzeugt. Ist der referierte Typ ein A Typ mit unbeschränktem Indexbereich oder ein REC Typ mit Diskriminanten ohne Diskriminantenfestlegung, so muß die Speicherbelegungsfunktion entweder einen expliziten Anfangswert (ein Ausdruck oder ein Aggregat) enthalten, oder einen Index oder eine Diskriminantenfestlegung. Dies ist der

einzige Fall, in dem ein Index oder eine Diskriminantenfestlegung erlaubt ist. Das erzeugte Objekt ist eingeschränkt entweder durch eine explizite Einschränkung oder durch die Schranken oder Diskriminanten der Anfangswerte.

Falls der referierte Typ oder einige seiner Komponenten vordefinierte Anfangswerte besitzen, werden diese Initialisierungen ausgeführt. Zum Schluß werden die explizit angegebenen Anfangswerte zugewiesen, wobei die Einschränkungen des referierten Typs zu beachten sind, und ein Zeigerwert, der auf das erzeugte Objekt zeigt, wird bereitgestellt. Das hiermit erzeugte Objekt bleibt solange erhalten, wie man potentiell darauf zugreifen kann. Andernfalls kann (aber muß nicht) der belegte Speicherplatz freigegeben werden (maschinenabhängig).

Beispiel:

Mit den Vereinbarungen

```
type zeiger;
type element is record daten : integer;
                       nachfolger : zeiger;
                  end record;
type zeiger is access element;
p : zeiger;
```

wird durch die Wertzuweisung

```
   p := new (element(10,p));
```

an die bestehende Liste mit Anfangsverweis p ein neues Element vorangestellt. Dieses Element ist mit dem Aggregat (10,p) initialisiert.

Hinweise:

(1) Wenn bei der Ausführung von new gegebene Einschränkungen unverträglich sind, erfolgt die Ausnahmebehandlung

constraint_error

(2) Wenn bei der Ausführung von new nicht genügend Speicherplatz für das zu erzeugende Objekt vorhanden ist, erfolgt die Ausnahmebehandlung storage_error.

(3) Wenn eine bestimmte Anwendung eine stärkere Kontrolle über die Speicherplatzbelegung bei Zeiger-Typen erforderlich macht, sind folgende Maßnahmen möglich:
   (a) Der gesamte Speicherplatzbedarf von Objekten eines Zeiger Typs kann durch eine Längenangabe festgelegt werden (siehe 8.2.1).
   (b) Mit der Direktive controlled (siehe 8.1) kann die automatische Speicherplatzrückgabe auf den Fall beschränkt werden, daß der Gültigkeitsbereich der Zeigertypvereinbarung verlassen wird.
   (c) Mit Hilfe des vordefinierten Programmschemas unchecked_deallocation kann der Speicherplatz einzelner Objekte vom referierten Typ freigegeben werden (siehe 8.3).

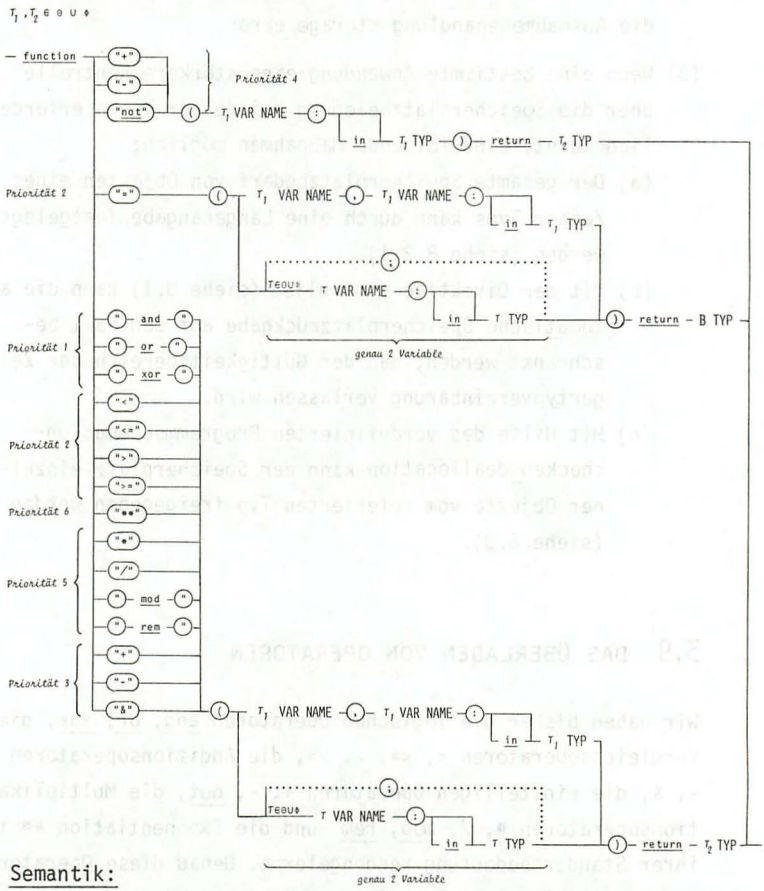
3.9 DAS ÜBERLADEN VON OPERATOREN

Wir haben bisher die logischen Operatoren and, or, xor, die Vergleichsoperatoren <, <=, >, >=, die Additionsoperatoren +, -, &, die einstelligen Operatoren +, -, not, die Multiplikationsoperatoren *, /, mod, rem und die Exponentiation ** in ihrer Standardbedeutung kennengelernt. Genau diese Operatoren können mittels einer Funktionsvereinbarung überladen werden, d.h. der entsprechende Operator kann beliebig vereinbarte Operandentypen zu einem beliebig vereinbarten Ergebnistyp verknüpfen. Die genaue Vorgehensweise wird in Abschnitt 5.7 ausführlich behandelt.

Syntax:

Aus

ADA 7: UNTERPROGRAMM SPEZIFIKATION (UPG SPEZIF)



Semantik:

Von den beiden Argumenten der Operatorvereinbarung entspricht das erste dem linken, das zweite dem rechten Operanden des Operators.

Der Operator "=" kann nur für limitierte geschützte Typen überladen werden. Mit dem Überladen von "=" ist implizit

auch "/=" überladen.

Haben ein lokaler und ein globaler Operator die gleichen Argument- und Ergebnistypen, so ist der lokale Operator gültig.

Hinweise:

(1) Der Operator "/=" darf nicht explizit überladen werden.

(2) Man beachte, daß das Überladen von Operatoren die Priorität nicht verändert.

3.10 DIE TYPKONVERTIERUNG

Mit Hilfe der Typkonvertierung kann man sich den strengen Typregeln, z.B. in Ausdrücken, bequem anpassen. Dabei erhöht der Zwang zu expliziten Typanpassungen die Durchsichtigkeit der durchgeführten Operationen.

Syntax:

Aus

ADA 31: τ OPERAND

— ZUN — τ_1 TYP NAME —(— τ_2 AUSDR —)—

Es sind folgende drei Fälle erlaubt:

(a) Konvertierung zwischen numerischen Typen:
Seien T_1 und T_2 beliebige numerische Typen. Dann wird der Wert des T_2 Ausdruckes in den Basistyp von T_1 umgewandelt. Bei real Typen liegt das Ergebnis innerhalb der vereinbarten Genauigkeit. Die Konvertierung von real nach integer erfordert i.a. eine Rundung.

(b) Konvertierung zwischen Feld Typen:
Seien T_1 und T_2 A TYPen. Die Konvertierung ist zulässig, wenn die Basistypen der Indexbereiche für jede Dimension gleich oder voneinander abgeleitete Typen sind und wenn die Komponententypen gleich oder voneinander abgeleitete Typen sind. Ist T_1 ein unbeschränkter A Typ, so sind die Grenzen des Ergebnisses die von T_2. Ist T_1 ein beschränkter A Typ, so müssen beide Typen zueinander passen. Die Grenzen des Ergebnisses sind die von T_1 und die Komponenten des Ergebnisses stimmen mit denen des Operanden überein.

(c) Konvertierung zwischen Basistyp und abgeleiteten Typen:
Die Konvertierung ist zulässig, wenn T_1 von T_2 bzw. T_2 von T_1 direkt abgeleitet ist. Diese Konvertierung kann Darstellungswechsel bewirken (siehe 8.2).

Beispiele:

zu (a): Die Konvertierung
integer(13.4)
konvertiert die Zahl 13.4 nach 13 oder 14 (maschinenabhängig).

zu (b):
```
type real is new float range 0 .. 30;
type a is array (1 .. 10) of float;
type b is array (19 .. 40) of real;
x : a;
```
Die Konvertierung
b(x)
konvertiert das Feld x in den Typ b mit den Indexgrenzen 19 .. 28 . (Basistyp ist hier integer).

Hinweise:

(1) Eine Typkonvertierung bewirkt die Ausnahmebehandlung constraint_error falls der Wert des Operanden einer Beschränkung des Ergebnistyps nicht genügt. Bei Typen bezieht sich das auch auf alle Indexfestlegungen.

3.11 ATTRIBUTE

Attribute bezeichnen bestimmte vordefinierte Eigenschaften von Größen. Dabei kann ein Attribut einen Wert, eine Funktion oder einen Typ oder Untertyp darstellen. Ein Attribut wird mit dem Namen der Größe, einem nachfolgenden Hochkomma und dem abschließenden Attributnamen (evtl. mit Argumentliste) bezeichnet.

In dieser Form treten sie auf in ADA 22: *T* TYP, in ADA 62: *T* ATTRIBUTIERTER FUNKTIONSNAME, in ADA 35: *T* ATTRIBUTIERTER FUNTIONSAUFRUF und in ADA 36 bis ADA 40: *T* KONSTANTE, bzw. in den Abschnitten 3.1 bis 3.5 und 9.8 .

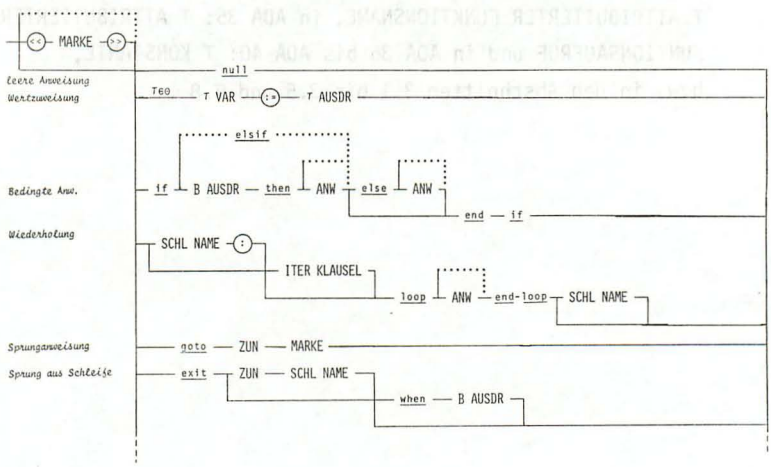
4. ANWEISUNGEN

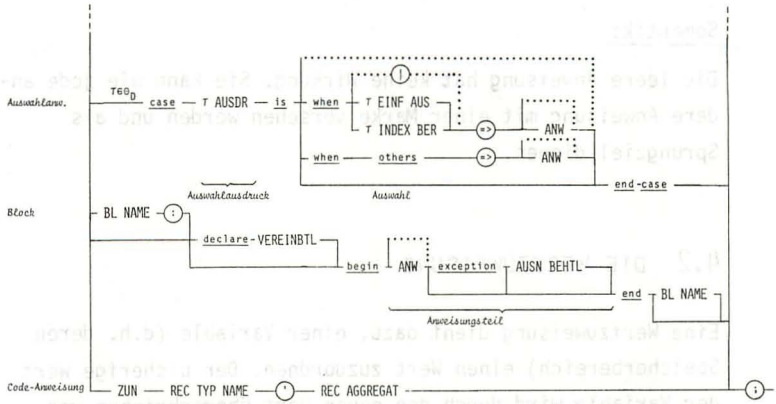
In diesem Abschnitt werden einige einfache Anweisungen eingeführt. Diese sind im *Anweisungsteil* einer Programmeinheit beliebig verwendbar, wobei natürlich nur die sinnvolle Anordnung auch einen sinnvollen Programmablauf ergibt. Syntaktisch ist keine Reihenfolge festgelegt, jedoch ist die Bedeutung (Semantik) des Programmes von den verwendeten Anweisungen und deren Reihenfolge abhängig. Daher muß das zu bearbeitende Problem zuvor, wie in A.3 beschrieben, so aufbereitet werden, daß die Reihenfolge der zu programmierenden Anweisungen daraus hervorgeht.

In ADA wird jede Anweisung durch ein Semikolon abgeschlossen, während z.B. in PASCAL das Semikolon zwei Anweisungen trennt. An jeder Stelle, wo eine Anweisung stehen darf, darf auch eine Folge von Anweisungen stehen.

Folgende Anweisungen werden hier näher behandelt:

Aus ADA 23: ANWEISUNG (ANW) UND ADA 24: STRUKTURIERTE ANWEISUNG





Außerdem werden einfache Ein- bzw. Ausgabeanweisungen behandelt.

Die Anweisungen zum Aufruf von Unterprogrammen, zum Auslösen einer Ausnahme oder zur Synchronisierung von Prozessen werden in den entsprechenden Kapiteln besprochen.

4.1 DIE LEERE ANWEISUNG

Die *leere Anweisung* hat keine Wirkung. Sie kann benützt werden, wenn syntaktisch eine Anweisung gefordert wird, aber vom Programmierer keine Wirkung bei der Ausführung des Programmes beabsichtigt ist.

Syntax:

Aus

ADA 23: ANWEISUNG (ANW)

Semantik:

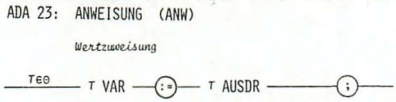
Die leere Anweisung hat keine Wirkung. Sie kann wie jede andere Anweisung mit einer Marke versehen werden und als Sprungziel dienen.

4.2 DIE WERTZUWEISUNG

Eine Wertzuweisung dient dazu, einer Variable (d.h. deren Speicherbereich) einen Wert zuzuordnen. Der bisherige Wert der Variable wird durch den neuen Wert überschrieben und geht damit verloren. Ab diesem Zeitpunkt trägt die Variable den neuen Wert.

Syntax:

Aus



Semantik:

Eine Wertzuweisung bewirkt die Auswertung des rechts stehenden *T* AUSDRUCKes mit dem Ergebnistyp ω und die Auswertung der links stehenden Variablen. Diese Auswertung kann parallel erfolgen. Kommt es jedoch bei Arrayzuweisungen zu Überlappungen, so werden zuerst die rechts stehenden Komponenten ausgewertet.

Dann erfolgt die Zuweisung von ω an die links stehende Variable, falls ω alle für die links stehende Variable geltenden Einschränkungen erfüllt.

Bei Arrayzuweisungen muß die Anzahl der Komponenten und der Komponententyp übereinstimmen.

Hinweise:

(1) Das Zuweisungszeichen := ist ein Zeichen, obwohl auf Eingabetastaturen in der Regel die beiden Tasten für : und = gedrückt werden müssen. Diese müssen jedoch unmittelbar aufeinanderfolgen. Zum Beispiel ist die Zeichenfolge :␣= kein Zuweisungszeichen.

(2) Der Wert ω ist bestimmt, wenn alle im rechts stehenden Ausdruck auftretende Größen einen bestimmten Wert haben (sonst unbestimmt). Nach Ausführung der Wertzuweisung hat dann die links stehende Variable ebenfalls einen bestimmten (sonst unbestimmten) Wert.

(3) Das Zuweisungszeichen := darf keineswegs mit dem Gleichheitszeichen = einer mathematischen Gleichung verwechselt werden. So ist die Wertzuweisung

```
i := i+1
```

durchaus sinnvoll (der Wert von i wird um 1 erhöht), während die Gleichung

```
i = i+1
```

mathematisch falsch ist.

(4) Ist der referierte Typ einer Zeigervariable ein Record Typ mit Diskriminanten, so können die Diskriminanten durch eine Wertzuweisung nicht verändert werden, da ja bei der Zeigertypvereinbarung der referierte Typ durch eine aktuelle Diskriminantenliste festgelegt wurde.

4.3 EINFACHE ANWEISUNGEN FÜR EIN- UND AUSGABE

Die Wertzuweisung als Mittel, einer Variable einen Anfangswert zuzuordnen, ist i.a. nicht flexibel genug. So läßt es sich beispielsweise mit der Wertzuweisung alleine ohne un-

sinnigen Schreibaufwand nicht realisieren, ein und dasselbe Programm mit verschiedenen Datensätzen zu rechnen. Dazu benötigt man eine Eingabeanweisung.

Andererseits genügt es uns nicht, wenn nach Beendigung des Programmes irgendwelche Variable des Programmes das Ergebnis enthalten, sondern wir wollen das Ergebnis auch sehen, z.B. auf Papier gedruckt oder am Bildschirm. Dazu benötigt man eine Ausgabeanweisung.

In der Regel erfordert jedes ADA-Programm ein auf die verwendeten speziellen Typen zugeschneidertes Ein-/Ausgabe-Paket. Zur Erstellung solcher Pakete stehen die Standardpakete low_level_io, input_output und text_io zur Verfügung. Das Paket low_level_io beschreibt Möglichkeiten, mit einem physikalischen Peripheriegerät direkt von einem ADA-Programm aus zu kommunizieren. Das Paketschema input_output beschreibt die Handhabung von Dateien (Files) mit Komponenten eines Typs und die Ein- und Ausgabe auf solche Files. Während hier die Daten in interner Form auf die Datei abgespeichert werden, beschreibt das Paket text_io die Handhabung von Textfiles. Das sind Dateien, auf denen Komponenten verschiedener Typen in lesbarer Form abgespeichert werden. Die Standardein- und ausgabe etwa von einem Bildschirmgerät aus wird stets mit solchen Files arbeiten. Im Paket text_io finden sich Pakete bzw. Paketschemata für die Ein- und Ausgabe aller skalaren Typen und für Strings. Ausprägungen der Schemata für die Standardtypen integer und float haben wir in dem Paket Standard_EA zusammengefaßt.

```
with text_io; use text_io;
package Standard_EA is
   package int_io is new integer_io (integer);
   package flo_io is new float_io (float);
   use  int_io; float_io;
end package Standard_EA;
```

Das Paket Standard_EA stellt alle Ein/Ausgabeunterprogramme des Pakets text_io zur Verfügung, also speziell die Vereinbarung von Textfiles, die Ein/Ausgabe für character, string und boolean sowie Funktionen und Prozeduren zur Gestaltung des Druckbilds.

Außerdem werden durch Erzeugung der Pakete int_io und float _io Ein/Ausgaberoutinen für die Standardtypen integer und float bereitgestellt.

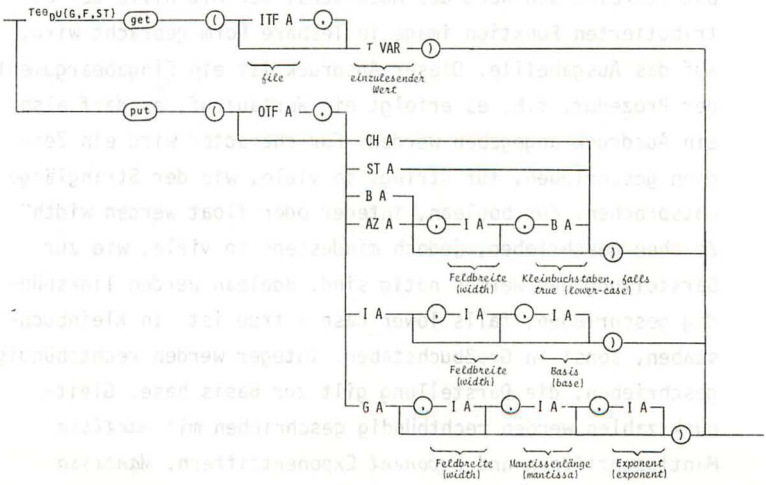
Wird also das Paket Standard_EA mittels einer Kontextangabe zu einer Programmeinheit gebunden, so kann in dieser Programmeinheit Ein- und Ausgabe von Standardtypen erfolgen. Die einfachsten und wichtigsten der Unterprogramme stellen wir hier vor. Eine genaue Beschreibung der gesamten Ein/Ausgabe und Filehandhabung erfolgt in Kapitel 11.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF

θ_D = {B,CH,AZ,I}



Als Kommentare stehen die Namen der formalen Argumente.

Semantik:

Es gelten folgende Vorbesetzungen der Argumente:

```
file = standard input bei get
     = standard output bei put
width = Ø
lower_case = false
base = 1Ø
manissa = float'digits
exponent = 2
```

Bei get wird eine Konstante des angegebenen Typs vom Eingabefile gelesen und mittels der attributierten Funktion value der angegebenen Variable zugewiesen. Man beachte, daß die eingelesene Variable ein Ausgabeargument von get ist. (Hier besteht die Gefahr einer Verwirrung: Der Begriff Ausgabeargument (Ausgangsargument) bezieht sich auf die Übergabeart des Arguments der Prozedur get. Die Prozedur get besorgt die Eingabe (das Einlesen) von Werten.)

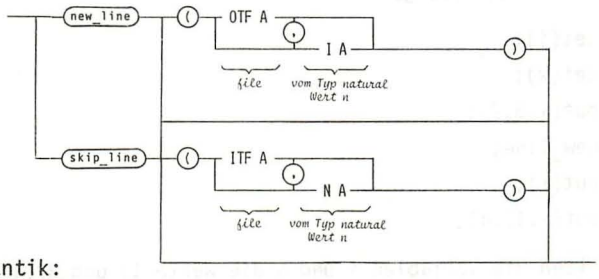
put schreibt den Wert des Ausdrucks, der mit Hilfe der attributierten Funktion image in lesbare Form gebracht wird, auf das Ausgabefile. Dieser Ausdruck ist ein Eingabeargument der Prozedur, d.h. es erfolgt ein Wertaufruf, es darf also ein Ausdruck angegeben werden. Für character wird ein Zeichen geschrieben, für strings so viele, wie der Stringlänge entsprechen. Für boolean, integer oder float werden width Zeichen geschrieben, jedoch mindestens so viele, wie zur Darstellung des Wertes nötig sind. Boolean werden linksbündig geschrieben, falls lower_case = true ist in Kleinbuchstaben, sonst in Großbuchstaben. Integer werden rechtsbündig geschrieben, die Darstellung gilt zur Basis base. Gleitpunktzahlen werden rechtbündig geschrieben mit *mantissa* Mantissenziffern und *exponent* Exponentziffern. *Mantissa* sollte nicht größer als float'digits sein, da nur so viele

Ziffern relevant sind.

Syntax:

Aus

ADA 53: FILEVERWALTUNGSPROZEDURAUFRUF



Semantik:

Textfiles sind zeilenorientiert.

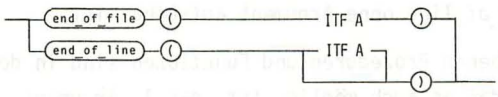
new_line beendet die Ausgabe einer Zeile auf das Ausgabefile (standard output) und gibt n-1 Leerzeilen aus (Vorbesetzung n = 1).

skip_line beendet die Eingabe einer Zeile vom Eingabefile (Vorbesetzung: standard input) und überliest n-1 weitere Zeilen (Vorbesetzung n = 1).

Syntax:

Aus

ADA 55: B E/A STANDARDFUNKTIONSAUFRUF



Semantik:

Das Ergebnis eines Aufrufs ist true, falls das Ende des Eingabefiles bzw. das Ende einer Zeile des Eingabefiles (standard input) erreicht ist, sonst false.

Beispiel:

Auf der Eingabedatei seien die folgenden Konstanten die nächsten zu lesenden Werte:

```
13    37.4
```

Durch die Anweisungen

```
get(i);
get(x);
put(i,5,2);
new_line;
put(i);
put(x,12,4);
```

erhalten die Variablen i und x die Werte 13 und 37.4 (evtl. durch Konvertierung verfälscht) und es werden folgende Zeichenfolgen gedruckt:

```
␣11Ø1
13␣␣␣3.74Ø0e+Ø1
```

notwendige Stellenzahl — angegebene Stellenzahl = 12

Hinweis:

Wir werden bis zum Kapitel 11 die Ein/Ausgabeprozeduren nur für die Standard-Ein-Ausgabedateien benutzen.

Man beachte, daß end_of_file stets ein Argument benötigt, während end_of_line ohne Argument aufrufbar ist.

Alle angegebenen Prozeduren und Funktionen sind in der Weise überladen, daß es auch möglich ist, das 1. Argument (das File) wegzulassen. Tatsächlich existieren z.B. für den Typ integer 2 verschiedene put Prozeduren, eine mit Angabe des Files, die andere ohne. Die Fortlassung der Formatangabe *width* oder der Basisangabe *base* erfolgt dagegen nach den in

ADA üblichen Aufrufregeln für Argumente mit vorbesetztem Anfangswert (vgl. Kap. 5).

4.4 DIE BEDINGTE ANWEISUNG

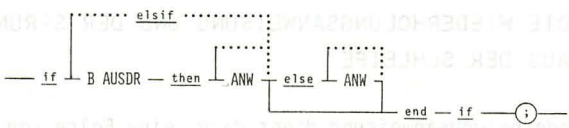
Soll ein Programm nicht nur stur einen einzigen möglichen linearen Ablauf haben, so bedarf es sogenannter V e r z w e i g u n g e n . Eine Verzweigung kann auf mehrere Arten erreicht werden. Wir besprechen hier die bedingte Anweisung, welche eine solche Verzweigung in Abhängigkeit vom logischen Aussagewert von logischen Ausdrücken ermöglicht.

Syntax:

Aus

ADA 24: STRUKTURIERTE ANWEISUNG

bedingte Anweisung



Semantik:

Die bedingte Anweisung

<u>if</u> β_1 <u>then</u> α_1 <u>elsif</u> β_2 <u>then</u> α_2 ... <u>elsif</u> β_n <u>then</u> α_n
<u>else</u> α_{n+1}

<u>end</u> <u>if</u>

mit den logischen Ausdrücken $\beta_1,\ldots,\beta_n$ und den Anweisungsfolgen $\alpha_1,\ldots,\alpha_{n+1}$ ermöglicht die Auswahl einer Anweisungsfolge α_i in Abhängigkeit von den logischen Ausdrücken β_j. Es werden nacheinander die logischen Ausdrücke $\beta_1,\beta_2,\ldots$ ausgewertet. Sobald sich für ein β_k der Wert true ergibt, wird die Anweisungsfolge α_k ausgeführt und die Ausführung

der bedingten Anweisung ist beendet; sind alle β_i false, wird α_{n+1} ausgeführt und die Ausführung der Anweisung beendet.

Beispiel:

```
if abso <= Ø.Ø1 then gut:= gut + 1;
elsif abso <= Ø.Ø5 then brauchbar:= brauchbar + 1;
elsif abso <= Ø.2 then schlecht:= schlecht + 1;
                  else unbrauchbar:= unbrauchbar + 1;
```

Hinweise:

Fehlen die mit elsif eingeleiteten Alternativen, so liegt die aus PASCAL bekannte bedingte Anweisung vor mit folgenden Ausnahmen:

Die einzelnen Alternativen sind Anweisungsfolgen.

Die bedingte Anweisung wird durch end if abgeschlossen.

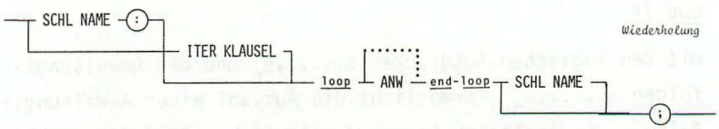
4.5 DIE WIEDERHOLUNGSANWEISUNG UND DER SPRUNG AUS DER SCHLEIFE

Die Wiederholungsanweisung dient dazu, eine Folge von Anweisungen solange zu wiederholen, bis eine Bedingung erfüllt ist oder eine feste Anzahl von Wiederholungen erreicht ist.

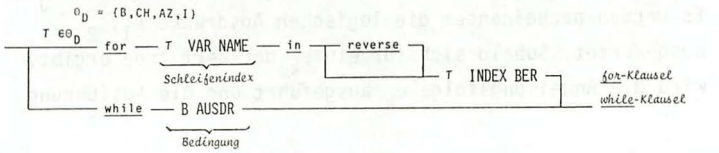
Syntax:

Aus

ADA 24: STRUKTURIERTE ANWEISUNG



ADA 25: ITERATIONSKLAUSEL (ITER KLAUSEL)



Semantik:

Ohne die Iterationsklausel wird die Anweisungsfolge solange ausgeführt, bis eine exit, return, goto Anweisung aus dieser Folge herausführt, eine Ausnahme auftritt oder die Ausführung durch eine Terminierungsanweisung abgebrochen wird.

Im Fall einer while-Klausel wird die Bedingung vor jedem Schleifendurchlauf geprüft und die Schleife nur durchlaufen, falls die Bedingung den Wert true hat.
Im Fall einer for-Klausel wird zuerst der Schleifenindex als lokale Variable eingeführt. Falls der Indexbereich leer ist, wird die Anweisungsfolge nicht durchlaufen. Sonst wird sie für jeden Wert des Indexbereiches einmal durchlaufen. Vor jedem Durchlauf wird der Wert an den Schleifenindex in aufsteigender (bei reverse in absteigender) Reihenfolge zugewiesen. Der Schleifenindex ist innerhalb der Anweisungsfolge eine Konstante, darf also nicht durch Wertzuweisungen oder Argumentübergabe vom Modus out oder in out verändert werden.

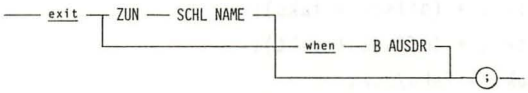
Der Schleifenname muß am Schluß erscheinen, falls er am Anfang steht.

Der Sprung aus Schleifen:

Die exit-Anweisung ist eine Anweisung zum Verlassen einer Schleife.

Syntax:

Aus ADA 23: ANWEISUNG



Semantik:

Ist ein Schleifenname angegeben, so wird die benannte Schlei Schleife, sonst die innere Schleife verlassen. Falls eine Bedingung angegeben ist, wird die Schleife nur verlassen, falls die Bedingung erfüllt ist.

Eine exit-Anweisung darf nur innerhalb einer Wiederholungsanweisung stehen.

Die Nachfolgeanweisung darf nicht außerhalb des Rumpfes stehen, der die Schleife enthält. Gegebenenfalls ist hier eine leere Anweisung einzufügen.

Beispiele:

1.
```
loop
    get(x);  abso:=abs(x-s);
    if abso <= Ø.Ø1
    then gut:= gut + 1;
    elsif abso <= Ø.Ø5
    then brauchbar:= brauchbar + 1;
    elsif abso <= Ø.2
    then schlecht:= schlecht + 1;
    else unbrauchbar:= unbrauchbar + 1;
    end if;
    i:= i + 1;
    exit when i = n;
end loop;
```

2.
```
while i  = 324
loop
    s:= s + (offset + faks);
    t:= t + (offset + fakt);
    faks:= faks/2.Ø;
    fakt:= fakt * 2.Ø;
    i:= i + 1;
end loop;
```

```
3. for i in 1 .. n
   loop
      get(x);
      xx:= x;
      qs:= Ø;
      while x < Ø
      loop
         qs:= qs + x mod 1Ø;
         x:= x/1Ø;
      end loop;
      put("Die Zahl"); put(xx,8);
      put("hat die Quersumme"); put(qs,5);
      new_line;
   end loop;
```

Hinweise:

Eine Wiederholungsanweisung mit for-Klausel beschreibt eine übliche Zählschleife mit der Schrittweite 1 oder -1.

Mittels einer while-Klausel oder exit-Anweisung kann man ferner jede beliebige Schleifenstruktur realisieren.

Bei einer Zählschleife ist der Schleifenindex lokale Größe, also außerhalb der Wiederholungsanweisung unbekannt.

4.6 MARKEN UND DIE SPRUNGANWEISUNG

Eine Anweisung kann mit einer oder mehreren Marken versehen sein. Solche Marken sind automatisch am Ende des Vereinbarungsteils des innersten Rumpfes, in dem sie auftreten, vereinbart.

Syntax:

Aus

ADA 23: ANWEISUNG

Semantik:

Das Setzen einer Marke vor einer Anweisung erhält erst Bedeutung im Zusammenhang mit einer entsprechenden Sprunganweisung. Insbesondere wird die Wirkung der Anweisung durch das Setzen der Marke nicht verändert. Beim Setzen einer Marke ist zu beachten, daß eine Marke nur einmal in ihrem Gültigkeitsbereich zum Markieren einer Anweisung verwendet werden darf.

Hinweis:

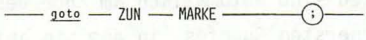
Im Gegensatz zu PASCAL gibt es keinen Markenvereinbarungsteil.

Eine Sprunganweisung dient, wie der Name sagt, zum Springen an eine andere Anweisung des Programmes. Sie unterbricht also die bisher gewohnte sequentielle Ausführung der Anweisungen eines Programmes.

Syntax:

Aus

ADA 23: ANWEISUNG



Semantik:

Die Sprunganweisung goto μ bewirkt, daß die Ausführung des

Programmes mit der markierten Anweisung <<μ>>α fortgesetzt wird. (Die Marke μ muß also die Anweisung markieren).

Sprünge von außen in eine strukturierte Anweisung oder einen Ausnahmebehandlungsteil sind ebenso verboten wie Sprünge zwischen verschiedenen Zweigen einer bedingten Anweisung, Auswahlanweisung oder select-Anweisung. Es darf nicht von einem Ausnahmebehandlungsteil zu einem anderen oder zu dem aufrufenden Block oder Rumpf zurückgesprungen werden. Außerdem darf nicht aus einem Rumpf oder einer accept-Anweisung herausgesprungen werden.

Es ist zu beachten, daß die Sprunganweisung goto μ nur im Gültigkeitsbereich der Marke μ auftreten darf.

Hinweise:

(1) Bei Verwendung von Sprunganweisungen stimmt die statische Anweisungsfolge (der Niederschrift) nicht mehr mit der dynamischen Anweisungsfolge (bei der Ausführung) überein. Solche Programme sind deshalb
- schwer zu lesen und zu verstehen
- schwer zu korrigieren
- bei Korrekturen schwerfällig.

(2) Die Verwendung von Sprunganweisungen ist wegen der zu beachtenden Blockstruktur und wegen der Unsicherheiten bei strukturierten Anweisungen möglichst zu vermeiden.

(3) Die Sprunganweisung kann mittels der Wiederholungsanweisung stets umgangen werden, was wegen (1) und (2) anzuraten ist.

4.7 DER UNTERPROGRAMMRÜCKSPRUNG

Durch eine Rücksprunganweisung (return-Anweisung) innerhalb eines Unterprogrammes wird die Ausführung des Unterprogram-

mes beendet und in die aufrufende Programmeinheit zurückgesprungen. (Siehe 5.2).

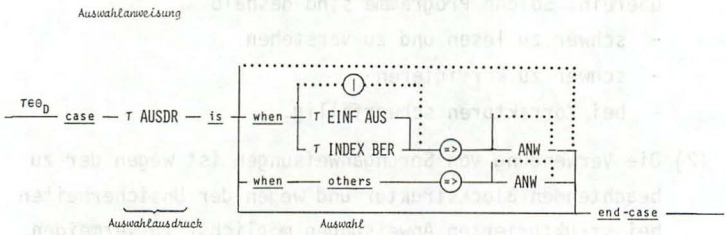
4.8 DIE AUSWAHLANWEISUNG (case-ANWEISUNG)

Die Auswahlanweisung kann als Verallgemeinerung der bedingten Anweisung (siehe Hinweis (1)) angesehen werden. Sie ermöglicht die Auswahl einer unter mehreren möglichen Anweisungen (Alternativen) in Abhängigkeit vom Wert eines Ausdruckes. Es kann also die auszuführende Alternative sozusagen berechnet werden. Damit stellt die Auswahlanweisung eine Verzweigung in eine vom Programmierer frei wählbare Anzahl von Richtungen dar.

Syntax:

Aus

ADA 24: STRUKTURIERTE ANWEISUNG



Semantik:

In Abhängigkeit vom Wert des Auswahlausdruckes wird genau eine der nachfolgend aufgelisteten Anweisungsfolgen ausgeführt. Dazu muß jeder Wert des Auswahltyps *T* in genau einer Auswahl auftreten. Die Ausdrücke in den Auswahlen sind statisch. Gibt es einen others-Zweig, so faßt dieser alle nicht vorkommenden Werte zusammen.

Mehrere Auswahlen vor einer Anweisung

```
when μ1 | μ2 | ... | μn => α;
```

sind äquivalent zu

```
when μ1 => α;
when μ2 => α;
  :
when μn => α;
```

oder, falls der Indexbereich μ1 .. μn genau die Werte μ1,.., μn enthält, äquivalent zu

```
when μ1 .. μn => α;
```

Beispiele:

1.
```
case quad is
    when 1 => f:= phi * r;
    when 2|4 => f:= 90 * r;
    when 3 => f:= -(phi - 180) * r;
end case;
```

2.
```
case ch is
    when 'a'|'A' => a:= a + 1;
    when 'e'|'E' => e:= e + 1;
    when 'i'|'I' => i:= i + 1;
    when 'o'|'O' => o:= o + 1;
    when 'u'|'U' => u:= u + 1;
    when others  => z:= z + 1;
end case;
```

Hinweise:

(1) Die bedingte Anweisung

```
if β then α1; else α2; end if;
```

ist äquivalent der case-Anweisung

```
case β is when true => α1;
          when false => α2;
end case;
```

(2) Die Auswahlanweisung in PASCAL läßt als Auswahl nur Konstanten zu.

(3) In anderen höheren Programmiersprachen wie BASIC, FORTRAN oder ALGOL 60 findet man eine ähnliche Anweisung unter dem Stichwort "berechnete Sprunganweisung".

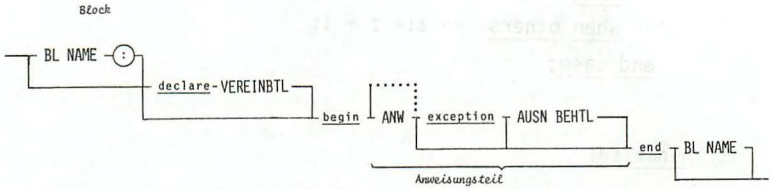
4.9 DER BLOCK

Ein Block faßt eine Folge von Anweisungen zu einer einzigen zusammen. Ein Block kann einen Vereinbarungsteil und einen Ausnahmebehandlungsteil enthalten. Blöcke dienen als Strukturierungsmittel, indem logisch zusammenhängende Anweisungen und die Vereinbarung der lokalen Typen, Konstanten und Variablen sowie die Behandlung von Ausnahmen zu einer Einheit zusammengefaßt werden.

Syntax:

Aus

ADA 24: STRUKTURIERTE ANWEISUNG



Semantik:

Falls ein Blockname benutzt wird, so muß er am Anfang und Ende stehen. Der Vereinbarungsteil wird zuerst verarbeitet und dann der Anweisungsteil ausgeführt.

Beispiel:

Der Block

```
fakultaet : declare
            i,fak_i : integer := 1;
            begin
            while i < n
            loop i:= i + 1;
                 fak_i:= fak_i * i;
            end loop;
            fak:= fak_i;
end fakultaet;
```

berechnet n! und weist diesen Wert der globalen Größe fak zu; n ist dabei eine globale Größe.

Hinweise:

(1) Die Verwendung eines Blockes entspricht dem Aufruf einer argumentlosen Prozedur.

(2) Blöcke können ineinander geschachtelt werden (vgl. Kap. 10).

(3) Die Verwendung von Blöcken entspricht derjenigen in ALGOL 60.

(4) Blöcke können zur Vereinbarung und Aktivierung von Prozessen verwendet werden.

4.10 DER AUFRUF EINER PROZEDUR

Nach Ersetzen der formalen durch die aktuellen Argumente wird der Prozedurrumpf ausgeführt (siehe 5.3).

4.11 ANWEISUNGEN ZUR SYNCHRONISIERUNG VON PROZESSEN

Zur Synchronisierung von Prozessen dienen vor allem der Eingangsaufruf, der eine Dienstleistung in einem anderen Prozeß aufruft, und die accept-Anweisung, die diesen Aufruf entgegennimmt. Diese beiden Anweisungen können auch als bedingter oder verzögerter Eingangsaufruf und als alternative accept-Anweisung formuliert werden.

Ein Prozeß kann durch eine Verzögerungsanweisung in den Wartezustand versetzt werden und durch eine bedingte oder vorzeitige Terminierung beendet werden.

Alle diese Anweisungen dürfen nur in Prozeßrümpfen oder deren Vorgänger auftreten. Eine genaue Beschreibung folgt in Kapitel 9.

4.12 DAS AUSLÖSEN EINER AUSNAHME

Durch eine raise-Anweisung wird eine Ausnahme ausgelöst und in den entsprechenden Ausnahmebehandlungsteil gesprungen (siehe Kapitel 7).

4.13 DIE CODE ANWEISUNG

Eine Code Anweisung ist ein qualifizierter Ausdruck, bestehend aus einem Aggregat eines Record Typs, der eine in Maschinencode geschriebene Anweisung definiert. Solche Anweisungen können nur innerhalb einer inline-Prozedur auftreten, die nur Code-Anweisungen und im Vereinbarungsteil nur use-

Klauseln und Pragmas enthält. Diese Konstruktion dient dazu, Maschinencode in ein ADA-Programm einzubinden (siehe 8).

4.14 BEISPIELE

Beispiel 1: Bilden einer Quersumme

Auf dem Datenteil stehen die Anzahl n und anschließend n natürliche Zahlen. Das folgende Programm liest zunächst die ganze Zahl n und nacheinander die natürlichen Zahlen ein und druckt mit erläuterndem Text jeweils deren Quersumme aus.

```
with standard_ea; use standard_ea;
procedure beispiel1 is
   n,qs,x,xx : integer;
   begin
      get(n);
      xx:= x;
      qs:= Ø;
      while x > Ø
      loop
         qs:= qs + x mod 1Ø;
         x:= x/1Ø;
      end loop;
      put("Die Zahl"); put(xx,8);
      put("hat die Quersumme"); put(qs,5);
      new_line;
   end beispiel1;
```

Beispiel 2: Maschinenaddition (Rundungsfehler)

Das Ergebnis einer Summation hängt beim Rechnen auf Rechenanlagen von der Reihenfolge der auszuführenden Additionen ab.

Das folgende Programm berechnet die Summe

$$S = \sum_{\nu=1(1)324} (2^{-29} + 2^{-\nu})$$

einmal durch Summation "von links nach rechts" und ein zweites Mal von "rechts nach links", also

$$T = \sum_{\nu=324(-1)1} (2^{-29} + 2^{-\nu})$$

Ausgedruckt werden die Werte S,T und die Differenz |S - T|.

```
with standard_ea; use standard_ea;
procedure beispiel2 is
   type real is digits 8;
   offset : real := 2.Ø ** (-29);
   fakt : real := 2.Ø ** (-324);
   faks : real := Ø.5;
   s,t : real := Ø.Ø;
   i : integer := 1;
   begin
      while i <= 324
      loop
         s:= s + (offset + faks);
         t:= t + (offset + fakt);
         faks:= faks/2.Ø;
         fakt:= fakt * 2.Ø;
         i:= i + 1;
      end loop;
      put("Summe S="); put(s,12); new_line;
      put("Summe T="); put(t,12); new_line;
      put("Differenz="); put(abs(s-t),12); new_line;
   end beispiel2;
```

Beispiel 3: (Meßwertaufgabe)

Auf dem Datenteil stehen n Meßwerte x_i, i = 1(1)n, für eine

Größe s. Die Beurteilung eines Meßwertes erfolge nach folgender Tabelle:

$\lvert x_k - s \rvert > 0.2$	"unbrauchbar"
$0.2 \geq \lvert x_k - s \rvert > 0.5_{10}-1$	"schlecht"
$0.5_{10}-1 \geq \lvert x_k - s \rvert > 0.1_{10}-1$	"brauchbar"
$0.1_{10}-1 \geq \lvert x_k - s \rvert$	"gut"

Folgendes Programm liest zunächst eine ganze Zahl n und den Wert s ein und druckt sodann von n einzulesenden Meßdaten die Anzahl der unbrauchbaren, schlechten, brauchbaren und guten Meßwerte gemäß obiger Tabelle aus.

```
with standard_ea; use standard_ea;
procedure beispiel3 is
   type real is digits 8;
   n,i,unbrauchbar, schlecht,brauchbar,gut : integer := 0;
   s,x,abso : real;
   begin
      get(n); get(s);
      loop
         get(x);
         abso:= abs(x - s);
         if abso <= 0.01
         then gut:= gut + 1;
         elsif abso <= 0.05
         then brauchbar:= brauchbar + 1;
         elsif abso <= 0.2
         then schlecht:= schlecht + 1;
         else unbrauchbar:= unbrauchbar + 1;
         end if;
         i:= i + 1;
         exit when i = n;
      end loop;
      put("Es wurden"); put(n);
```

```
      put_line("Messwerte eingelesen");
      put("Davon waren"); put(gut); put_line("gut");
      put("            "); put(brauchbar);
      put_line("brauchbar");
      put("            "); put(schlecht);
      put_line("schlecht");
      put(             "); put(unbrauchbar);
      put_line("unbrauchbar");
   end beispiel3;
```

Beispiel 4: Suche von Extremwerten

standard_funktionen sei ein vordefiniertes Bibliothekspaket, das u.a. die Funktion sqrt zur Berechnung der Quadratwurzel aus positiven Zahlen vom Typ real beinhaltet.

```
with standard_ea,standard_funktionen;
use standard_ea,standard_funktionen;
procedure beispiel4 is
   type real is digits 8;
   max,min,absmax,absmin,x,absx,xx,y,ymax : real;
   i,n : integer;
   begin
      get(n); get(x);
      max:= x; min:= x;
      absmax:= abs(x); absmin:= absmax;
      ymax:= Ø;
      i:= 2;
      while i <= n
      loop
         xx:= x; get(x); absx:= abs(x);
         if x > max
         then max:= x;
         elsif x < min
         then min:= x;
```

```
        end if;
        if absx > absmax
        then absmax:= absx;
        elsif absx < absmin
        then absmin:= absx;
        end if;
        y:= sqrt(abs(x - xx));
        if y > ymax
        then ymax:= y;
        end if;
        i:= i + 1;
    end loop;
    put("Groesster Wert max          ="); put(max);
    new_line;
    put("Kleinster Wert min          ="); put(min);
    new_line;
    put("Betragsgroesster Wert absmax="); put(absmax);
    new_line;
    put("Betragskleinster Wert absmin="); put(absmin);
    new_line;
    put("Folgenmaximum ymax          ="); put(ymax);
    new_line;
  end beispiel4;
```

Beispiel 5: Polynomauswertung (Hornerschema) und Tabellierung

Ein Polynom fünften Grades $p(x) = a_5x^5+a_4x^4+a_3x^3+a_2x^2+a_1x+a_0$ soll im Intervall [a;b] an den Stützstellen x_i ausgewertet werden; dabei ist $h = \frac{b-a}{n}$, $x_0 := a$ und $x_{i+1} := x_i + h$, $i = 0,\ldots,n-1$. Der Wert des Polynoms an der Stelle x berechnet sich nach folgendem Schema (HORNER-Schema)

$$p(x) = ((((a_5x+a_4)x+a_3)x+a_2)x+a_1)x+a_0$$

Das folgende Programm liest die Koeffizienten $a_0, a_1, \ldots, a_5$ sowie a,b und n ein und druckt die Stützstellen x_i, i=Ø(1)n, und die zugehörigen Polynomwerte sowie diejenigen Stützpunkte aus, an denen der Polynomwert minimal bzw. maximal wird.

```
with standard_ea; use standard_ea;
procedure beispiel5 is
   type real is digits 8;
   aØ,a1,a2,a3,a4,a5,a,b,h,x,y,xmax,ymax,xmin,ymin : real;
   i,n : integer;
   begin
      get(aØ);get(a1);get(a2);get(a3);get(a4);get(a5);
      get(a);get(b);get(n);
      h:= (b - a)/n;
      -- Tabellenüberschrift
      put_line("  x(i)        y(x(i))  ");
      x:= a;
      -- Hornerschema für p(a)
      y:= ((((a5*x+a4)*x+a3)*x+a2)*x+a1)*x+aØ;
      put(x); put(y); new_line
      ymax:= y; ymin:= y; xmax:= x; xmin:= x;
      i:= Ø;
      loop
         x:= x + h;
         -- Hornerschema für p(a+i*h)
            ...((a5*x+a4)*x+a3)*x+a2)*x+a1)*x+aØ;
         -- Tabellierung
         put(x); put(y); new_line
         -- Extremabestimmung
         if y > ymax
         then ymax:= y;
         elsif y < ymin
         then ymin:= y;
```

```
        end if;
        i:= i + 1;
        exit when i = n;
    end loop;
    put_line("Maximum:"); put(xmax); put(ymax); new_line;
    put_line("Minimum:"); put(xmin); put(ymin); new_line;
end beispiel5;
```

Beispiel 6: Vokale zählen

Das folgende Programm bestimmt die Anzahl der einzelnen Vokale (groß und klein geschrieben) in einem Text. Es werden solange neue Zeichen des Textes eingelesen, bis das Ende des Datenteils erreicht ist, in dem der Text steht. Dies kann in ADA durch den booleschen Standardfunktionsaufruf end_of_file(input) geprüft werden. Ausgedruckt werden dann für jeden Vokal die gefundene Anzahl und schließlich die Zahl der sonstigen Zeichen.

```
with standard_ea; use standard_ea;
procedure beispiel6 is
   a,e,i,o,u,z : integer := Ø;
   ch : character;
   begin
      loop
         get(ch);
         case ch is
            when 'a'|'A' => a:= a + 1;
            when 'e'|'E' => e:= e + 1;
            when 'i'|'I' => i:= i + 1;
            when 'o'|'O' => o:= o + 1;
            when 'u'|'U' => u:= u + 1;
            when others  => z:= z + 1;
         end case;
         exit when end_of_file(input);
```

```
    end loop;
    put_line("   Anzahl der Vokale   ");
    put_line("   A   E   I   O   U   ");
    put(a); put(e); put(i); put(o); put(u);
    new_line;
    put(z); put_line("  sonstige Zeichen");
  end beispiel6;
```

Beispiel 7:

Das folgende Programm tabelliert unter Verwendung der Auswahlanweisung die mit den Polarkoordinaten r und ϕ definierte Funktion $f(\phi,r)$ in der komplexen Zahlenebene $\mathbb{C}$ auf dem Ursprungskreis mit Radius $\bar{r}$, beginnend mit dem Winkel $\bar{\phi} = 0$ in der Schrittweite h. Radius und Schrittweite werden eingelesen.

$$f(\phi,r) := \begin{cases} \phi \cdot r & 0 \leq \phi < 90 \;,\; r \geq 0 \\ 90 \cdot r & 90 \leq \phi < 180 \;,\; r \geq 0 \\ -(\phi - 180)\cdot r & 180 \leq \phi < 270 \;,\; r \geq 0 \\ -90 \cdot r & 270 \leq \phi < 360 \;,\; r \geq 0 \end{cases}$$

```
with standard_ea; use standard_ea;
procedure tabelle is
quad : integer;
f,r,h,phi : float;
begin
   get(h); get(r); phi:= 0;
   while phi <= 360
   loop quad:= integer(phi/90) + 1;
        case quad is
           when 1 => f:= phi * r;
           when 2 => f:= 90 * r;
           when 3 => f:= -(phi - 180) * r;
           when 4 => f:= -90 * r;
        end case;
```

```
        put("radius="); put(r);
        put("winkel="); put(phi);
        put("funktionswert="); put(f);
        phi:= phi + h;
    end loop;
end;
```

5. HANDHABUNG VON UNTERPROGRAMMEN

Ein UNTERPROGRAMM ist eine ausführbare Programmeinheit, welche durch einen Unterprogrammaufruf angestoßen, d.h. zur Ausführung gebracht wird. Es besteht aus der UNTERPROGRAMM SPEZIFIKATION und dem *Unterprogrammrumpf*.

Die Unterprogrammspezifkation beschreibt die Schnittstelle des Unterprogramms zu anderen Programmeinheiten; in ihr sind also alle für den Aufruf wichtigen Informationen zusammengefaßt.

Im einzelnen sind dies der Name des Unterprogramms sowie die formale Argumentliste. Beim Aufruf werden die formalen Argumente durch die aktuellen ersetzt. Die aktuellen Argumente dienen dazu, Daten in das Unterprogramm einzubringen oder Ergebnisse herauszureichen.

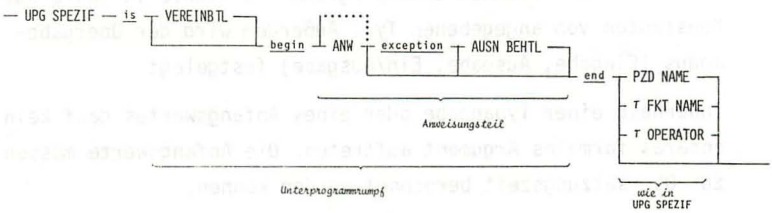
Der Unterprogrammrumpf beschreibt die Aktion des Unterprogramms, d.h. die auszuführende Anweisungsfolge. Ein Unterprogramm wird immer dann verwendet werden, wenn die gleiche Anweisungsfolge mehrfach - eventuell mit verschiedenen Daten - ausgeführt werden soll. Doch aus Gründen der Lesbarkeit, Übersicht und besseren Strukturierung eines Programmes wird empfohlen, zusammengehörige Anweisungsfolgen stets als Unterprogramm zu formulieren, auch wenn sie nur einmal ausgeführt werden, z.B. Initialisierung oder Ausdrucken von Ergebnissen.

5.1 DIE VEREINBARUNG VON UNTERPROGRAMMEN

Die Vereinbarung von Unterprogrammen wurde in 1.1 einführend behandelt. Wir wiederholen hier kurz.

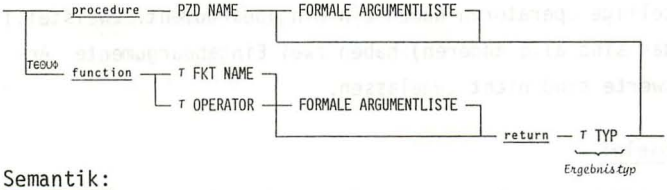
Syntax:

ADA 6: UNTERPROGRAMM (UPG)



ADA 7: UNTERPROGRAMM SPEZIFIKATION (UPG SPEZIF)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}



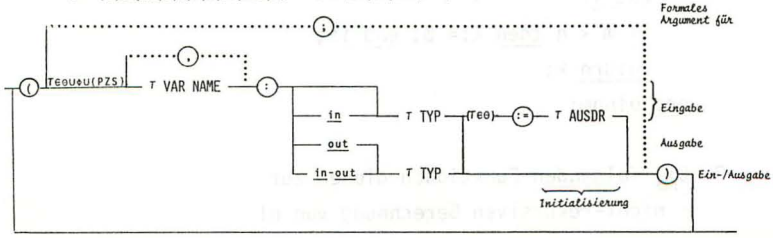
Semantik:

Die Verarbeitung einer UNTERPROGRAMMSPEZIFIKATION wird, auch wenn diese wiederholt wird, nur einmal vorgenommen. Sie beginnt mit der Einführung des Namens, es folgt die Verarbeitung der formalen Argumente in der Reihenfolge ihres Auftretens und bei Funktionen oder Operatoren die Verarbeitung des Ergebnistyps.

Syntax:

ADA 27: FORMALE ARGUMENTLISTE (FOR ARG LISTE)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Φ = {IF,OF,IOF,ITF,OTF}



Semantik:

Die Namen der formalen Argumente werden eingeführt. Diese gelten in dem gesamten Unterprogramm als lokale Variable oder Konstanten vom angegebenen Typ. Außerdem wird der Übergabemodus (Eingabe, Ausgabe, Ein/Ausgabe) festgelegt.

Innerhalb einer Typangabe oder eines Anfangswertes darf kein anderes formales Argument auftreten. Die Anfangswerte müssen zur Übersetzungszeit berechnet werden können.

Funktionen dürfen nur Eingabeargumente haben.

Prozesse können nur als Eingabeargumente auftreten.

Einstellige Operatoren haben ein Eingabeargument. Zweistellige (das sind alle anderen) haben zwei Eingabeargumente. Anfangswerte sind nicht zugelassen.

Beispiele:

1. Funktionsvereinbarung aus 5.8, Beispiel 2

```
function binom (m,n : in integer) return integer is
i,k : integer := 1;
begin
   if (m >= n) and (n > Ø)
      then while i < n + 1 loop
               k:= (k * (m - i + 1))/i;
               i:= i + 1;
            end loop;
   end if;
   if m < n then k:= Ø; end if;
   return k;
end binom;
```

2. Die folgenden Funktionen dienen zur
   - nicht-rekursiven Berechnung von n!

```
function fak (n : in integer) return integer is
i,fak_i : integer := 1;
begin
   while i < n loop
      i:= i + 1;
      fak_i:= fak_i * i;
   end loop;
   return fak_i;
end fak;
```

- rekursiven Berechnung von n!

```
function rek_fak (n : in integer) return integer is
begin
   if n = Ø then return 1;
            else return rek_fak (n-1) * n;
   end if;
end rek_fak;
```

3. Die Prozedur halb berechnet die Nullstelle ξ der Funktion $f(x) = x^2 - c1$ im Intervall $[a_1,b_1]$, $Ø \leq a_1 \leq b_1$. Beginnend mit dem Ausgangsintervall $[a_1,b_1]$, das ξ enthält, wird das jeweils berechnete Intervall solange verkleinert (halbiert), bis der Durchmesser kleiner als eps1 ist.

```
procedure halb (c1,eps1 : float, a1,b1 : in out float) is
m : float;
begin
   loop
      m:= (a1 + b1) * Ø.5;
      if m * m - c1 >= Ø then b1:= m;
                         else a1:= m;
      end if;
      exit when  b1 - a1 < eps1;
   end loop;
end halb;
```

4. Operatorvereinbarung für Vektoraddition

```
function "+" (x,y : vektor) return vektor is
resultat : vektor(x'range);
begin
   for index in x'range
   loop
      resultat(index):= x(index) + y(index);
   end loop;
   return resultat;
end "+";
```

Hinweise:

Die formalen Eingabeargumente werden innerhalb des Unterprogrammes wie lokale Konstante, die anderen wie lokale Variable behandelt. Ihre Namen dürfen also nicht im Vereinbarungsteil des Unterprogrammrumpfes überschrieben werden.

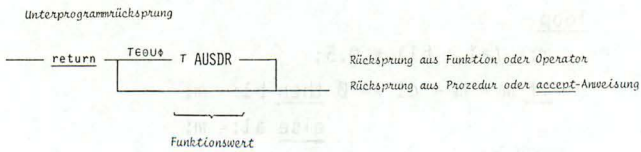
Der Übergabemechanismus wird in 5.6 beschrieben.

5.2 DER UNTERPROGRAMMRÜCKSPRUNG

Syntax:

Aus

ADA 23: ANWEISUNG (ANW)



Semantik:

- Durch einen *Unterprogrammrücksprung* (Rücksprunganweisung,

return-Anweisung) innerhalb eines Unterprogramms wird die Ausführung des Unterprogramms beendet und in die aufrufende Programmeinheit zurückgesprungen.

- Der Typ des Funktionswertes muß gleich dem Ergebnistyp der Funktion sein.
- Die Ausführung einer Prozedur wird auch korrekt beendet, falls das Ende des Anweisungsteils erreicht ist, eine Funktion muß dagegen mit einer Rücksprunganweisung verlassen werden, da nur dadurch der Funktionswert festgelegt wird. Wird bei einer Funktion das Ende des Anweisungsteils erreicht, so ist der Wert der Funktion unbestimmt.
- Eine Rücksprunganweisung kann innerhalb einer inneren Schleife eines Unterprogramms stehen, alle Schleifen werden dann korrekt beendet und der Rücksprung erfolgt.
- Eine Rücksprunganweisung darf nur im Anweisungsteil eines Unterprogrammrumpfes oder einer accept-Anweisung stehen. Dort kann sie selbstverständlich mehrfach auftreten.

Beispiel:

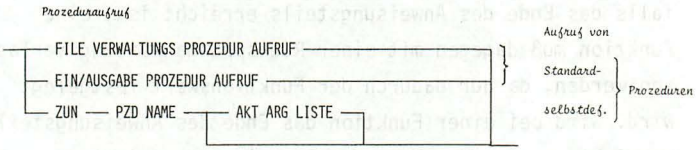
```
return;    -- Ruecksprung aus einer Prozedur
return (3 + i) * x;    -- Ruecksprung aus einer Funktion
                -- Als Funktionswert wird der Wert des Aus-
                -- drucks (3 + i)x übergeben.
```

5.3 DER AUFRUF EINER PROZEDUR

Syntax:

Aus

ADA 23: ANWEISUNG (ANW)



Der Aufruf von Standardprozeduren zur Fileverwaltung und Ein/Ausgabe hat die gleiche Form wie der Aufruf selbstdefinierter Prozeduren. Die genaue Behandlung der Standardprozeduren erfolgt in Kapitel 11.

Semantik:

Der Aufruf einer Prozedur bewirkt die Ersetzung der formalen durch die aktuellen Argumente (siehe 5.6) und die Ausführung des entsprechenden Prozedurrumpfes.

Beispiel:

halb(2,1.Øe-5,Ø,2);

berechnet $\sqrt{2}$ mit der Genauigkeit $1Ø^{-5}$.

5.4 DER AUFRUF EINER FUNKTION

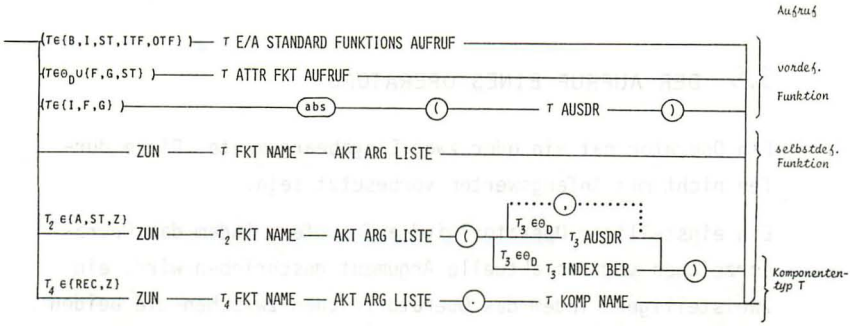
Der Aufruf einer Funktion erfolgt innerhalb eines Ausdrucks.

Syntax:

Aus

ADA 31: T OPERAND (T OPD)

T Funktionsaufruf



Der Aufruf von Standardfunktionen für Ein- und Ausgabe wird in Kapitel 11 behandelt. Die attributierten Funktionen sowie die Standardfunktion abs wurden in Kapitel 3 bei den Ausdrücken für die einzelnen Typen behandelt.

Semantik:

- Die Auswertung des Ausdrucks wird unterbrochen. Die formalen Argumente werden durch die aktuellen ersetzt und das Ergebnis gemäß dem Funktionsaufruf bestimmt.
- Bei Funktionen von strukturiertem Ergebnistyp kann auf die Komponenten zugegriffen werden wie bei Variablen.

Beispiel:

Die Anweisung

```
m:= binom (n-t,s-k);
```

weist m den Wert $\binom{n-t}{s-k}$ zu.

Der Aufruf

```
f:= rek_fak (2 * j - 1);
```

bewirkt für j = 2 die Zuweisung des Wertes 6 an f.

Hinweis:

Eine Funktion berechnet einen Wert und sollte keine Nebenwirkungen auf globale Größen haben.

5.5 DER AUFRUF EINES OPERATORS

Ein Operator hat ein oder zwei Eingabeargumente. Diese dürfen nicht mit Anfangswerten vorbesetzt sein.

Ein einstelliger Operator wird aufgerufen, indem das Operatorzeichen vor das aktuelle Argument geschrieben wird, ein zweistelliger, indem das Operatorzeichen zwischen die beiden aktuellen Argumente geschrieben wird.

In Kapitel 3 (Ausdrücke) wurden die vordefinierten Operatoren besprochen und die Syntax für das allgemeine Operatorkonzept dargestellt.

Beispiel:

Die Anweisung

```
c:= a + b;
```

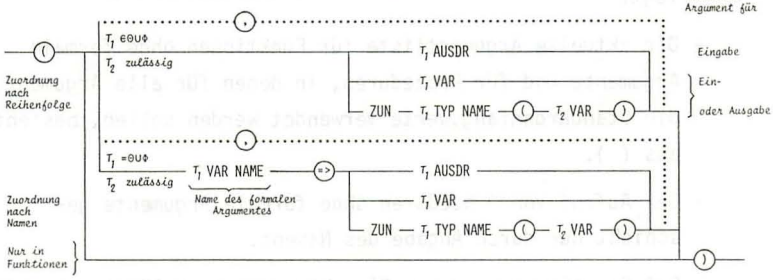
berechnet mit den 3 Vektoren a,b,c die Summe a + b und weist das Ergebnis an c zu. Dabei wurde die Operatorvereinbarung für den Operator + von 5.1, Beispiel 4 vorausgesetzt.

5.6 DAS ZUSAMMENSPIEL DER FORMALEN UND DER AKTUELLEN ARGUMENTE

Syntax:

ADA 34: AKTUELLE ARGUMENTLISTE (AKT ARG LISTE)

θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}



Semantik:

- Die *aktuellen Argumente* werden den formalen in der Reihenfolge von links nach rechts zugeordnet (*Zuordnung nach Reihenfolge*), oder sie ersetzen das formale Argument, dessen Name angegeben wurde (*Zuordnung nach Namen*).
- Hat ein formales Argument einen Anfangswert (default) (nur bei Eingabeargumenten), so kann das aktuelle Argument hierfür fehlen, es wird dann mit dem Anfangswert gerechnet.
- Fehlt aus diesem Grund ein aktuelles Argument oder wurde einmal die Ersetzung durch Angabe des formalen Argumentnamens durchgeführt, so müssen alle weiteren Argumente durch Angabe des formalen Argumentnamens ersetzt werden.
- Aktuelle Ein/Ausgabe- oder Ausgabeargumente sind Variable oder Typkonversionen von Variablen.
- Die aktuellen Argumente müssen vor dem Aufruf alle Einschränkungen der formalen erfüllen. Für Ausgabeargumente für skalare Typen und Zeigertypen genügt es, wenn die

formalen Argumente die Einschränkungen der aktuellen Argumente vor dem Rücksprung erfüllen.

- Die Auswertung der Ausdrücke oder Komponentenauswahl etc. erfolgt vor der Ausführung des Rumpfes.
- Die Reihenfolge der Argumentauswertung ist nicht festgelegt.
- Die aktuelle Argumentliste für Funktionen ohne formale Argumente und für Prozeduren, in denen für alle Argumente die Standardanfangswerte verwendet werden sollen, besteht aus ( ).
- Der Aufruf von Prozeduren ohne formale Argumente geschieht nur durch Angabe des Namens.
- Bei Funktionen sind nur Eingabeargumente zulässig, es müssen eine oder mehrere Rücksprung-(<u>return</u>-) Anweisungen vorkommen (s. 5.2). Wird die Funktion nicht durch eine Rücksprunganweisung verlassen, so ist der Wert undefiniert.
- Aktuelle Eingabeargumente vom limitierten, geschützten Typ können auch Prozesse sein.
- Eingabeargumente fungieren als lokale Konstante des Unterprogrammes, deshalb darf ihr Wert innerhalb des Rumpfes nicht verändert werden. Für skalare Typen und Zeigertypen wird der Wert des aktuellen Argumentes in das formale Argument kopiert.
- Ausgabeargumente (Argumente für Variable) dienen dazu, Ergebnisse aus einer Prozedur zu liefern. Für skalare Typen oder Zeigertypen wird der Wert des formalen Argumentes beim Prozedurrücksprung in das aktuelle Argument kopiert. Bei Typkonvertierung für Variable wird vor der Ergebnisrückgabe die inverse Typkonvertierung durchgeführt.
- Für strukturierte Typen ist der Übergabemechanismus für

Ein- und Ausgabe nicht festgelegt.

5.7 DAS ÜBERLADEN VON UNTERPROGRAMMEN

Namen von Unterprogrammen (also auch Operatoren) können mehrfach belegt werden. Ein Unterprogramm wird also durch seinen Namen, die Anzahl, Reihenfolge, Namen und Typen seiner Argumente, das Vorhandensein von Anfangswerten für spezielle Argumente und bei Funktionen auch durch den Ergebnistyp identifiziert. Unterscheiden sich deshalb zwei Unterprogramme nur in mindestens einem dieser Punkte, so sind sie unterscheidbar und können daher beide gleichzeitig benutzt werden. Beim Aufruf solcher Unterprogramme muß eindeutig feststellbar sein, welches gemeint ist. Sonst liegt ein Fehler vor. Unterscheiden sich z.B. zwei Unterprogramme nur durch den Namen der formalen Argumente, so müssen zur eindeutigen Identifizierung die aktuellen Argumente mit dem Namen des formalen Argumentes angegeben werden.

Beispiel:

Die Operatorvereinbarung von 5.1, Beispiel 4 versieht den arithmetischen Standardoperator + mit einer zusätzlichen Bedeutung: Addition für Vektoren.

5.8 BEISPIELE

Beispiel 1:

In der folgenden Prozedur wird j! für j = 1,2,...,k berechnet.

```
with standard_ea;
procedure fakultaeten is
use standard_ea;
j,k : integer;
function fak(n : in integer) return integer
is i,fak_i : integer := 1;
   begin
      while i < n loop
         i:= i + 1;
         fak_i:= fak_i * i;
      end loop;
      return fak_i;
   end fak;
   begin
      get(k); j:= Ø;
      while j < k loop
         j:= j + 1;
         put(j);
         put(fak(j));
         new_line;
      end loop;
   end fakultaeten;
```

Beispiel 2:

Ein Kartenspiel mit n Karten werde so verteilt, daß jeder Spieler s Karten erhält. Befinden sich t Trümpfe im Kartenspiel, so berechnet sich die Wahrscheinlichkeit w(k), daß ein bestimmter Spieler genau k Trümpfe erhält, wie folgt:

$$w(k) = \frac{\binom{t}{k}\binom{n-t}{s-k}}{\binom{n}{s}}, \quad k = \emptyset(1)t .$$

Für j Datensätze n,t,s sind die Wahrscheinlichkeiten w(Ø), w(1),...,w(t) zu berechnen.

```
with standard_ea;
procedure trumpf is
use standard_ea;
j,k,n,s,t : integer;
w : float;

function binom (m,n : in integer) return integer is
i,n : integer := 1;
begin
   if (m >= n) and (n > Ø)
      then while i < n + 1 loop
                k:= (k * (m - i + 1))/i;
                i:= i + 1;
            end loop;
   end if;
   if m < n then k:= Ø; end if;
   return k;
end binom;

begin          -- Hauptprogramm
   get(j);
   while j > Ø loop
      get(n);
      get(t);
      get(s);
      put("n=");
      put(n);
      put("␣t="); put(t); put("␣s="); put(s); new_line(2);
      k:= Ø;
      while k <= t loop
```

```
        w:= real(binom(t,k)*binom(n-t,s-k)/binom(n,s));
        put("Wahrscheinlichkeit␣fuer␣"); put(k);
        put("␣Truempfe␣:␣");put(w);new_line;
        k:= k + 1;
      end loop;
      new_line;
      j:= j - 1;
    end loop;
end trumpf;
```

Beispiel 3:

Gegeben sei das Anfangswertproblem

$$y' = f(x,y),\ y(x_0) = y_0$$

im Intervall [a,b]. Wir wollen im folgenden Programm die Lösung $y(x)$ an den Stellen $x_j := a + j \cdot h$, $j = 0,1,\ldots,n$, mit $h := \frac{b-a}{n}$ näherungsweise bestimmen. Wir verwenden dazu nebeneinander drei numerische Verfahren:

Euler-Verfahren:

$$y_{j+1} := y_j + h \cdot f(x_j,y_j),\ j = 0,1,\ldots,n-1$$

Verbessertes Euler-Verfahren:

$$x_{j+1} := y_j + \frac{h}{2}(f(x_j,y_j) + f(x_j+h, y_j+h\cdot f(x_j,y_j))),$$
$$j = 0,1,\ldots,n-1$$

Runge-Kutta-Verfahren:

$$y_{j+1} := y_j + \frac{h}{6}(k_1+4k_2+k_3) \text{ mit}$$
$$k_1 := f(x_j,y_j)$$
$$k_2 := f(x_j+\frac{h}{2}, y_j+\frac{h}{2}k_1)$$
$$k_3 := f(x_j+h, y_j-hk_1+2hk_2)$$
$$j = 0,1,\ldots,n-1 \quad .$$

Dabei bedeutet y_j jeweils die durch das Verfahren berechnete Näherung für den Funktionswert $y(x_j)$.

Die Verfahren werden für $f(x,y) = e^{-y}$ durchgeführt und die Näherungswerte mit den entsprechenden Funktionswerten der exakten Lösung $y(x) = \ln x$ verglichen.

```
with standard_ea, standard_funktionen;
procedure beispiel3 is      -- numerische Loesung des Anfangs-
                            -- wertproblems
   use standard_ea, standard_funktionen;
   x,y,y0,y1,y2,y3,a,b,h,k1,k2,k3 : float;
   i,n : integer;
   function f (x,y : in float) return float is
   begin
      return exp(-y);
   end f;
begin     -- Beispiel 3
   get(a); get(b); get(y0); get(n);
   -- Ueberschrift
   put("␣␣␣␣␣x␣␣␣␣␣␣y1␣␣␣␣␣␣y2␣␣␣␣␣␣y3␣␣␣␣␣␣y(x)");
   y1:= y0;
   y2:= y0;
   y3:= y0;
   i:= 1;
   x:= a;
   h:= (b - a)/float(n);
   y:= ln(x);
   put(x,width => 6);
   put(y1,6);
   put(y2,6);
   put(y3,6);
   put(y,6);
   while i <= n loop
      -- Euler-Verfahren
      y1:= y1 + h * f(x,y1);
```

```
      -- Verbessertes Euler-Verfahren
      y2:= y2 + h/2 * (f(x,y2)+f(x+h,y2+h*f(x,y2)));
      -- Runge-Kutta-Verfahren
      k1:= f(x,y3);
      k2:= f(x + h/2,y3 + h/2 * k1);
      k3:= f(x+h,y3-h*k1+2*h*k2);
      y3:= y3 + h/6 * (k1 + 4 * k2 + k3);
      -- Berechnung des Arguments
      x:= x + h;
      -- Vergleichswert y(x)
      y:= ln(x);
      -- Tabellierung
      put(x,width => 6); put(y1,6); put(y2,6); put(y3,6);
      put(y,6);
      i:= i + 1;
   end loop;
end beispiel3;
```

6. HANDHABUNG VON PROGRAMMSCHEMATA

Unterprogramme beschreiben Anweisungsfolgen, die zur Laufzeit (Ausführungszeit) durch Aufruf mit verschiedenen aktuellen Argumenten ausgeführt werden. Als Argumente können Ausdrücke und Variable angegeben werden. Oft kommt es vor, daß gleiche Anweisungsfolgen für verschiedene Typen gebraucht werden, z.B. die Verwaltung einer Liste, deren Grundtyp eine beliebige, für die Listenverwaltung nicht interessierende Datenstruktur sein kann, oder Matrizenrechnung für reelle bzw. komplexe Matrizen. Oder man benötigt ein Unterprogramm, das man in gleicher Weise auf verschiedene Funktionen anwenden will, z.B. ein Integrationsverfahren.

Nicht nur Unterprogramme sondern auch Pakete sollten parametrisiert vorliegen, d.h. mit verschiedenen Argumenten belegt werden können.

Für diesen Zweck stehen in ADA die Schemata für Unterprogramme und Pakete zur Verfügung. Ein S c h e m a (eine generische Programmeinheit) ist dabei eine S c h a b l o n e , die eine ganze Klasse von ähnlichen Programmeinheiten (Unterprogrammen, Paketen) beschreibt.

Während der Übersetzungszeit können durch eine UNTERPROGRAMMERZEUGUNG oder PAKETERZEUGUNG verschiedene A u s p r ä g u n g e n (Inkarnationen) dieser Schablone hergestellt werden, die dann wie normale Unterprogramme oder Pakete behandelt werden.

Für verschiedene Aktionen sind Programmschemata vordefiniert. Von diesen Schemata können ohne vorherige Vereinbarungen Ausprägungen erzeugt werden. Diese Schablonen treten vor allem bei der Ein- und Ausgabe auf (siehe 4.3 und 11), sowie bei der unkontrollierten Speicherfreigabe (8.3) bzw. Typumwand-

lung (8.4) und bei der Verwaltung gemeinsamer Variablen von Prozessen (9.7).

6.1 DIE VEREINBARUNG VON SCHEMATA

Syntax:

ADA 3: UNTERPROGRAMM SCHEMA SPEZIFIKATION

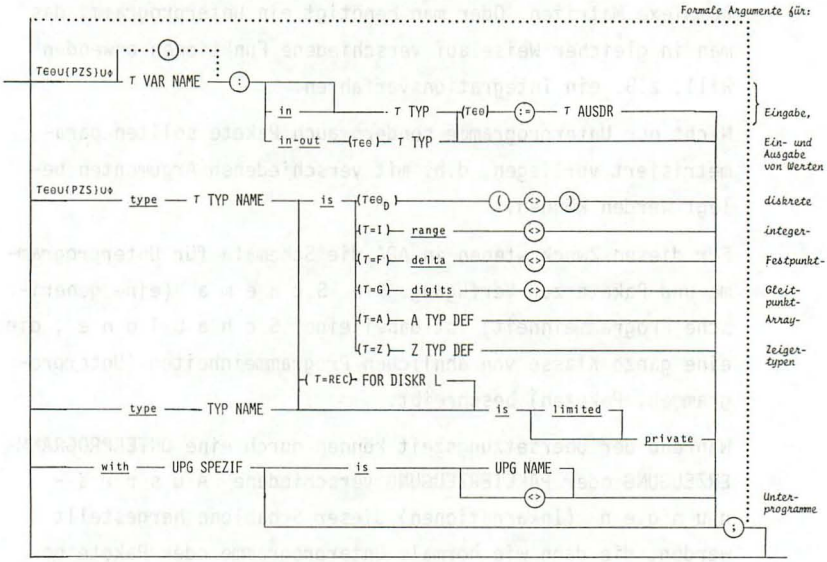
—— generic —— FOR SCH ARG LISTE —— UPG SPEZIFIKATION ——

ADA 4: PAKET SCHEMA SPEZIFIKATION

—— generic —— FOR SCH ARG LISTE —— PAK SPEZIFIKATION ——

ADA 5: FORMALE SCHEMA ARGUMENT LISTE (FOR SCH ARG LISTE)

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$ $\Phi = \{IF,OF,IOF,ITF,OTF\}$ $\Theta_D = \{B,CH,AZ,I\}$



Semantik:

- Für ein Unterprogramm, das als Schema vereinbart wird, müssen Spezifikation und UNTERPROGRAMM getrennt angegeben werden. Im Unterprogrammrumpf werden die formalen Schemaargumente verwendet. Bei der Unterprogrammerzeugung werden in der Spezifikation und im Rumpf die formalen Schemaargu-

mente durch aktuelle ersetzt.

- Besteht ein Paket, das als Schema vereinbart wird, aus Spezifikation und Rumpf, so können im Rumpf die formalen Schemaargumente der Spezifikation verwendet werden. Bei der Paketerzeugung werden in Spezifikation und Rumpf die formalen Schemaargumente durch aktuelle ersetzt.
- Bei der Verarbeitung einer Schemaspezifikation wird zuerst der Name des Schemas eingeführt, dann folgt die Verarbeitung der formalen Schemaargumente in der gegebenen Reihenfolge.
- Ein formales Schemaargument kann von einem anderen abhängen, falls dieses ein Typargument ist und vorher in der formalen Schemaargumentliste auftritt.
- Ausdrücke für Anfangswerte, die von formalen Schemaargumenten abhängen, werden erst während der Erzeugung ausgewertet.
- Formale Schemaargumente dürfen nicht als Auswahlmarken, in integer Typdefinitionen oder Genauigkeitseinschränkungen auftreten.
- Für formale Schemaargumente für Ein- und Ausgabe von Werten muß die Zuweisung möglich sein. (Also sind keine Files oder Prozesse erlaubt).
- Man beachte, daß keine reinen Ausgabeargumente erlaubt sind.
- Formale Schemaargumente für Typen werden verarbeitet wie Typvereinbarungen.
- Es stehen alle vordefinierten Operationen, Konstanten (auch attributierte) und attributierten Funktionen, außer image und value, zur Verfügung. Zum Beispiel werden Arrays in der üblichen Weise indiziert und für integer Typen stehen die Operationen +, -, *, / zur Verfügung.

Für geschützte Typen müssen alle Operationen über formale Schemaargumente für Unterprogramme zur Verfügung gestellt werden, nur Zuweisung und Vergleich auf Gleichheit oder Ungleichheit sind bekannt.

- Formale Schemaargumente für Unterprogramme können selbstverständlich formale Argumente haben, die auch von formalen Schemaargumenten für Typen abhängen können. Ist eine *Vorauswahl* angegeben, so wird bei der Erzeugung ein Unterprogramm dieses Namens bzw. ein beliebiges passendes (falls Vorauswahl <>) eingesetzt, falls ein aktuelles Schemaargument fehlt.
- Prozeßtypen können nicht als formale Schemaargumente auftreten, da durch Angabe eines Prozeßtyps alle Objekte dieses Prozeßtyps festliegen und ihren Wert nicht ändern können. Demzufolge könnte auch kein anderer Prozeßtyp als aktuelles Schemaargument übergeben werden.

Hinweise:

Wesentlich neu im Vergleich zu Unterprogrammen ist die Übergabe von Typen und Unterprogrammen als Argumente.

Bei den Schemaargumenten für Typen sollte man den Typ soweit wie möglich kennzeichnen, da dann die vordefinierten Operationen bekannt sind und während der Übersetzungszeit mögliche Prüfungen ausgeführt werden können.

Soll ein Schema für verschiedene Typen (Typklassen) angewandt werden, so müssen diese als geschützte Typen übergeben werden.

Es können nun auch Pakete parametrisiert werden. Innerhalb einer Schemavereinbarung oder dem entsprechenden Rumpf können g l o b a l e Größen auftauchen; das sind Größen, die in einer die Schemavereinbarung oder den Rumpf umfassenden Programmeinheit vereinbart wurden.

Ein Unterprogrammrumpf muß stets vorhanden sein. Man kann nicht zu einer Unterprogrammschemaspezifikation mehrere aktuelle Rümpfe schreiben und nur die Spezifikation anwenden.

Beispiele:

1. Zeigerverwaltung

   Das Paketschema zeiger_verwaltung stellt dem Benutzer die Möglichkeit der "garbage collection" zur Verfügung. Es wird der zeiger_typ definiert, mit dem die Komponenten einfach verkettet werden. Das Paket enthält die Prozeduren ablegen und holen, um ein überflüssig gewordenes Listenelement an die Liste anzuhängen bzw. ein Listenelement aus der Liste zu entnehmen. Erst wenn die Liste leer ist, wird in holen ein neues Listenelement erzeugt.

```
generic
   type komponente is private;
package zeiger_verwaltung is
   type komponenten_rahmen;
   type zeiger_typ is access komponenten_rahmen;
   type komponenten_rahmen is
      record
         daten : komponente;
         nachfolger : zeiger_typ;
      end record;
   procedure ablegen (zeiger : in out zeiger_typ);
   procedure holen (zeiger : out zeiger_typ);
end zeiger_verwaltung;

package body zeiger_verwaltung is
   listen_anfang : zeiger_typ := null;
   procedure ablegen (zeiger : in out zeiger_typ) is
```

```
begin
   zeiger.nachfolger:= listen_anfang;
   listen_anfang:= zeiger;
   zeiger:= null;
   exception
      when constraint_error => null;
            -- wird ausgeloest, wenn zeiger = null war,
            -- also ablegen mit einem unbelegten Zei-
            -- ger aufgerufen wurde.
end ablegen;
procedure holen (zeiger : out zeiger_typ) is
begin
   if listen_anfang = null
      then zeiger:= new komponenten_rahmen;
      else zeiger:= listen_anfang;
            listen_anfang:= listen_anfang.nachfolger;
            zeiger.nachfolger:= null;
      end if;
end holen;
end zeiger_verwaltung;
```

2. Horner-Schema für Polynome beliebigen Grades mit Koeffizienten beliebigen Typs

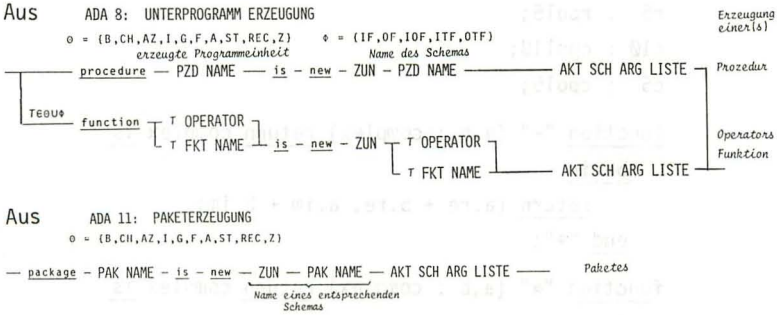
```
generic
   type koefftyp is private;
   type polynom is array (integer range <>) of koefftyp;
   with function "+" (a,b : koefftyp) return
         koefftyp is "+";
   with function "*" (a,b : koefftyp) return
         koefftyp is "*";
function horner (a:polynom; x : koefftyp) return koefftyp;
function horner (a:polynom; x : koefftyp) return koefftyp
   is z : koefftyp;
```

```
begin
    z:= a(a'last);
    for i in reverse a'first .. a'last - 1
    loop
        z:= z * x + a(i);
    end loop;
    return z;
end horner;
```

6.2 DIE AUSPRÄGUNG VON SCHEMATA

Ein Programmschema (Unterprogrammschema oder Paketschema) ist noch keine fertige direkt verwendbare Programmeinheit. Erst durch die während der Übersetzungszeit vorgenommene PAKETERZEUGUNG bzw. UNTERPROGRAMMERZEUGUNG werden weiterverwendbare Pakete bzw. Unterprogramme erstellt.

Syntax:



Semantik:

Der Name der erzeugten Programmeinheit wird eingeführt, anschließend wird die Ausprägung des Schemas vorgenommen, indem die formalen durch die aktuellen Schemaargumente ersetzt werden (s. 6.3). Dadurch liegt nun eine normale Programmein-

heit vor, deren Verarbeitung wie beschrieben (1.2 oder 5.1) vorgenommen wird.

Namen für globale Größen bezeichnen stets die Werte, die bei der Schemavereinbarung oder Formulierung des Rumpfes angesprochen wurden, nicht die bei der Erzeugung angesprochenen.

(vgl. PASCAL: globale Größen einer Prozedur)

Beispiel: Ausprägungen des Funktionsschemas horner

```
procedure horneraufrufe is
   type complex is record re,im : float; end record;
   type rpol10 is array (0 .. 10) of float;
   type rpol5  is array (integer range 0 .. 5) of float;
   type cpol10 is array (0 .. 10) of complex;
   type cpol5  is array (0 .. 5) of complex;
   -- Koeffizientenfelder für reelle und komplexe Polynome
   -- 10. und 5. Grades.
   r10 : rpol10;
   r5  : rpol5;
   c10 : cpol10;
   c5  : cpol5;

   function "+" (a,b : complex) return complex is
      begin
         return (a.re + b.re, a.im + b.im);
      end "+";

   function "*" (a,b : complex) return complex is
      begin
         return (a.re*b.re-a.im*b.im,a.re*b.im+a.im*b.re);
      end "*";

   -- Jetzt sind alle verwendeten Operationen bekannt.
   -- Erzeugung verschiedener Funktionen
   -- aus dem Schema horner
```

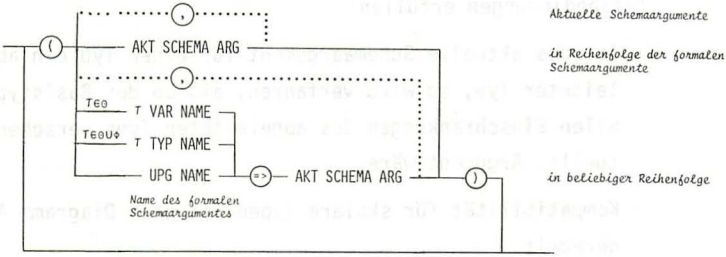
```
function rhorner1Ø is new horner (float,rpol1Ø);
   -- + und * als Standardvorbesetzung
function rhorner5 is new horner (float,rpol5,"+","*");
   -- + und * angegeben
function chorner1Ø is new horner (complex,cpol1Ø);
   -- + und * als Standardvorbesetzung
function chorner5 is new horner (complex,cpol5,
   komplexe_Arithmetik."+",komplexe_Arithmetik."*");
   -- + und * aus Paket komplexe_Arithmetik;
--Berechnung und Ausgabe der Werte.
```

6.3 DAS ZUSAMMENSPIEL DER FORMALEN UND DER AKTUELLEN SCHEMAARGUMENTE

Syntax:

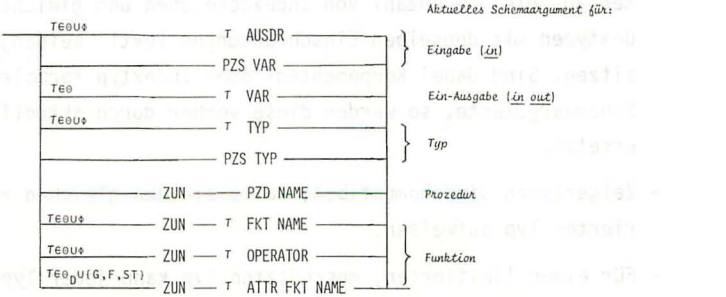
ADA 12: AKTUELLE SCHEMA ARGUMENT LISTE (AKT SCH ARG LISTE)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Φ = {IF,OF,IOF,ITF,OTF}



ADA 13: AKTUELLES SCHEMA ARGUMENT (AKT SCHEMA ARG)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Φ = {IF,OF,IOF,ITF,OTF} Θ_D = {B,CH,AZ,I}



Semantik:

Ersetzung der Argumente der Reihenfolge nach und durch Angabe des formalen Argumentnamens erfolgt nach den gleichen Regeln wie Ersetzung der formalen durch die aktuellen Argumente bei Unterprogrammaufrufen.

Die Ersetzung der Argumente für Eingabe von Werten erfolgt durch Kopieren des aktuellen Argumentes in das formale. Dieses tritt innerhalb des erzeugten Paketes bzw. Unterprogrammes wie eine Konstante auf.

Bei einem Prozeß bezeichnen das formale und aktuelle Argument den gleichen Prozeß.

Ein aktuelles Argument für Ein/Ausgabe muß eine Variable (also keine Typkonversion) von einem Typ sein, für den die Zuweisung verfügbar ist.

Aktuelle Argumente für Typen müssen folgende Kompatibilitätsbedingungen erfüllen:

- Ist das aktuelle Schemaargument für einen Typ ein abgeleiteter Typ, so wird verfahren, als ob der Basistyp, mit allen Einschränkungen des abgeleiteten Typs versehen, aktuelles Argument wäre.
- Kompatibilität für skalare Typen ist nach Diagramm ADA 5 geregelt.
- Array Typen sind kompatibel, falls sie gleichen Komponententyp, gleiche Anzahl von Indexbereichen und gleiche Indextypen mit denselben Einschränkungen (evtl. keinen) besitzen. Sind dabei Komponenten- oder Indextyp formale Schemaargumente, so werden diese vorher durch aktuelle ersetzt.
- Zeigertypen sind kompatibel, falls sie den gleichen referierten Typ aufweisen.
- Für einen limitierten, geschützten Typ kann jeder Typ

(auch Prozeßtyp oder Filetyp) übergeben werden.

- Für einen geschützten Typ muß Zuweisung und Vergleich auf Gleichheit für den aktuellen Typ zur Verfügung stehen.
- Tritt eine formale Diskriminantenliste auf, so muß sie für formalen und aktuellen Typ identisch sein.
- Unterprogramme, die als aktuelle Schemaargumente übergeben werden, müssen mit den formalen Unterprogrammen in Reihenfolge, Übergabeart und Typ (einschließlich Einschränkungen) der Argumente übereinstimmen, für Funktionen und Operatoren muß der Ergebnistyp übereinstimmen. Anwesenheit und Wert von Anfangswerten sind unerheblich (Das sind die gleichen Regeln wie bei der Umbenennung, siehe 2.8).
- Funktionen können als Operatoren übergeben werden und umgekehrt.

6.4 BEISPIELE

Beispiel 1:

Paketschema für Matrix- und Vektorarithmetik:

Als Schemaargumente werden übergeben:

Komponententyp der Matrizen/Vektoren

Nullelement des Komponententyps

Additions- und Multiplikationsoperatoren für den Komponententyp

Für Vektoren und Matrizen mit dynamischem Indexbereich stehen dann folgende Operatoren und Funktionen zur Verfügung:
(s eine Komponente; x,y Vektoren; a,b Matrizen)

s * x	Skalar · Vektor
x + y	Vektor + Vektor
x * y	Vektor · Vektor (Skalarprodukt)
trans(a)	Transponieren einer Matrix
s * a	Skalar · Matrix
a + b	Matrix + Matrix
a * b	Matrix · Matrix

```
generic
   type komponente is private;
   nullelement : komponente;
   with function "+"(x,y:komponente) return komponente
         is <>;
   with function "*"(x,y:komponente) return komponente
         is <>;
package vektoren_und_matrizen_paket is
   type vektor is array(integer range <>) of komponente;
   type matrix is array(integer range <>,integer range <>)
                  of komponente;
   function "*"(s:komponente;x:vektor) return vektor;
   function "+"(x,y:vektor) return vektor;
   function "*"(x,y:vektor) return komponente;
```

```
   function trans(a:matrix) return matrix;
   function "*"(s:komponente;a:matrix) return matrix;
   function "+"(a,b:matrix) return matrix;
   function "*"(a,b:matrix) return matrix;
   function "*"(a:matrix;x:vektor) return vektor;
   index_fehler,anschlusszahl_fehler : exception;
end vektoren_und_matrizen_paket;

package body vektoren_und_matrizen_paket is
   function "*"(s:komponente;x:vektor) return vektor is
      resultat : vektor(x'range);
      begin
         for index in x'range
         loop
            resultat(index):= s * x(index);
         end loop;
         return resultat;
      end "*";
   function "+"(x,y:vektor) return vektor is
      resultat : vektor(x'range);
      begin
         for index in x'range
         loop
            resultat(index):= x(index) + y(index);
         end loop;
         return resultat;
      end "+";
   function "*"(x,y:vektor) return komponente is
      resultat : komponente := nullelement;
      begin
         for index in x'range
         loop
            resultat:= resultat + x(index) * y(index);
         end loop;
         return resultat;
```

```
    end "*";
  function trans(a:matrix) return matrix is
    resultat : matrix(a'range(2),a'range(1));
    begin
      for zeile in a'range(2)
      loop
        for spalte in a'range(1)
        loop
          resultat(zeile,spalte):= a(spalte,zeile);
        end loop;
      end loop;
      return resultat;
    end trans;
  function "*"(s:komponente;a:matrix) return matrix is
    resultat : matrix(a'range(1),a'range(2));
    begin
      for zeile in a'range(1)
      loop
        for spalte in a'range(2)
        loop
          resultat(zeile,spalte):= s * a(zeile,spalte);
        end loop;
      end loop;
      return resultat;
    end "*";
  function "+"(a,b:matrix) return matrix is
    resultat : matrix(a'range(1),a'range(2));
    begin
      for zeile in a'range(1)
      loop
        for spalte in a'range(2)
        loop
          resultat(zeile,spalte):= a(zeile,spalte) +
                                   b(zeile,spalte);
        end loop;
```

```
      end loop;
      return resultat;
   end "+";
function "*"(a,b:matrix) return matrix is
   resultat : matrix(a'range(1),b'range(2));
   prod_sum : komponente;
   begin
      for zeile in a'range(1)
      loop
         for spalte in b'range(2)
         loop
            prod_sum:= nullelement;
            for ink in Ø .. a'length(2)-1
            loop
               prod_sum:= prod_sum
                          +a(zeile,a'first(2)+ink)
                          *b(b'first(1)+ink,spalte);
            end loop;
            resultat(zeile,spalte):= prod_sum;
         end loop;
      end loop;
      return resultat;
   end "*";
function "*"(a:matrix;x:vektor) return vektor is
   resultat : vektor(a'range(1));
   prod_sum : komponente;
   begin
      for index in a'range(1)
      loop
         prod_sum:= nullelement;
         for ink in Ø .. x'length-1
         loop
            prod_sum:= prod_sum
                       +a(index,a'first(2)+ink)
                       *x(x'first+ink);
```

```
            end loop;
            resultat(index):= prod_sum;
        end loop;
        return resultat;
    end "*";
end vektoren_und_matrizen_paket;
```

Beispiel 2:

Das Paketschema schlangenpaket stellt dem Benutzer ein Warteschlangenkonzept für einen beliebigen Datentyp zur Verfügung. Aus einer Schlange werden die Elemente in der Reihenfolge ihrer Ankunft wieder ausgegeben (FIFO). laenge gibt die Anzahl der Elemente in einer Schlange an. Der Versuch, einer leeren Schlange ein Element zu entnehmen, löst die Ausnahme schlange_leer aus (siehe 7.3). Intern wird eine Schlange durch den geschützten Typ schlangen_infos repräsentiert. Es wird das Paket zeiger_verwaltung benutzt (siehe 6.1, Beispiel 1).

```
with zeiger_verwaltung;
generic
   type element is private;
package schlangenpaket is
   type schlangen_infos is private;
   schlange_leer : exception;
   procedure push (schlange : in out schlangen_infos;
                   elem :  in element);
   procedure pop (schlange : in out schlangen_infos;
                   elem : out element);
   function laenge (schlange : in schlangen_infos) return
                   integer;
   private
      package listen_verwaltung is
         new zeiger_verwaltung (komponente => element);
      use listen_verwaltung;
```

```
      type schlangen_infos is
         record
            first,last : zeiger_typ := null;
            laenge     : integer := 0;
         end record;
end schlangenpaket;
package body schlangenpaket is
use listen_verwaltung;
   procedure push (schlange : in out schlangen_infos;
                   elem : in element) is
   begin
      if schlange.last = null
         then holen (schlange.last);
              schlange.last.all:= (elem,null);
              schlange.first:= schlange.last;
         else holen (schlange.last.nachfolger);
              schlange.last.nachfolger.all:= (elem,null);
              schlange.last:= schlange.last.nachfolger;
      end if;
      schlange.laenge:= schlange.laenge + 1;
   end push;

   procedure pop (schlange : in out schlangen_infos;
                  elem : out element) is
   begin
      if schlange.laenge = 0 then raise schlange_leer;
      end if;
      elem:= schlange.first.daten;
      aushaengen:
         declare
            hilfszeiger : zeiger_typ;
            begin
               hilfszeiger:= schlange.first;
               schlange.first:= schlange.first.nachfolger;
```

```
            ablegen (hilfszeiger);
        end aushaengen;
    schlange.laenge:= schlange.laenge - 1;
    if schlange.laenge = Ø then schlange.last:= null;
    end if;
  end pop;
  function laenge (schlange : in schlangen_infos) return
                integer is
  begin
    return schlange.laenge;
  end laenge;
end schlangenpaket;
```

7. DIE BEHANDLUNG VON AUSNAHMEN

Eine Ausnahme (exception) stellt bei der Ausführung eines Programmes eine Situation dar, in welcher die normale Ausführung des Programmes unmöglich wird. Dieser Fall kann eintreten aufgrund von Fehlern oder außergewöhnlichen Situationen während der Ausführung. Bei Eintreten einer speziellen Ausnahme werden nachfolgende Aktionen des Programmes ausgelöst, die wir unter dem Begriff A u s n a h m e b e h a n d l u n g zusammenfassen. Verschiedene Standardfehlersituationen lösen automatisch bei Ausführung des Programmes vordefinierte Ausnahmen aus, die zum Abbruch der Programmausführung führen (siehe 7.1). Diese Standardreaktionen werden nur dann ausgeführt, wenn vom Programmierer keine andere Behandlung vorgeschrieben ist.

Prinzipiell dienen drei Sprachelemente zur Ausnahmebehandlung:

- Durch eine entsprechende Vereinbarung wird der Name für eine Ausnahmesituation vereinbart.
- Mit der raise-Anweisung wird die Behandlung einer Ausnahme ausgelöst.
- Im Ausnahmebehandlungsteil, der am Ende des Anweisungsteils einer Programmeinheit steht, sind unter dem Namen der Ausnahme bzw. unter others die fehlerbehandelnden Aktionen festgelegt.

7.1 VORDEFINIERTE AUSNAHMEN

Die Ausnahmen constraint_error
numeric_error
select_error

storage_error
tasking_error

sind für jede Programmeinheit standardmäßig vordefiniert. Beim Auftreten einer entsprechenden Fehlersituation wird die Ausnahmebehandlung automatisch ausgelöst. Diese Fehlersituationen werden zur Laufzeit des Programmes durch entsprechende (durch Pragmas unterdrückbare) Tests abgefragt und führen dann zum Programmabbruch. Die vordefinierten Ausnahmen kommen also ohne Ausnahmevereinbarung, raise-Anweisung und Ausnahmebehandlungsteil zur Wirkung. Dies schließt jedoch nicht die explizite Auslösung durch eine raise-Anweisung aus. Weiter kann die Zuordnung eines eigenen Ausnahmebehandlungsteils erfolgen oder die vordefinierte Ausnahme durch eine Ausnahmevereinbarung überschrieben werden.

Die vordefinierten Ausnahmen werden in folgenden Situationen ausgelöst. Dabei ist zu beachten, daß gewisse interne Kontrollen zur Entdeckung von Fehlersituationen durch die Angabe eines entsprechenden Pragmas innerhalb eines Blocks bzw. des Rumpfes eines Unterprogrammes, Paketes oder Prozesses unterdrückt werden können (siehe ADA 52 und 8.1):

vordefinierte Ausnahme	unterdrückbarer interner Test	Fehlersituation
constraint_error	access_check	Versuch, über einen null-Verweis auf ein Objekt zuzugreifen
	discriminant_check	Versuch, eine nicht mit einer vorgegebenen Beschränkung verträgliche Diskriminante zuzuweisen oder auf eine Komponente zuzugreifen, die für die gegenwärtige Diskriminante nicht existiert

	index_check	Versuch, mit den Indexwerten den zulässigen Indexbereich zu verlassen
	length_check	Versuch, eine unzulässige Länge von Feldern oder Teilfeldern zu benutzen
	range_check	Versuch, einen Wert zuzuweisen, der außerhalb des geforderten Wertebereichs liegt bzw. eine unverträgliche Index-, Diskriminanten- oder Bereichseinschränkung zu verwenden
numeric_error	division_check	Bei Division oder Restbildung wird durch Null dividiert
	overflow_check	Ergebnis einer numerischen Operation liegt außerhalb des zulässigen Wertebereichs
storage_error	storage_check	Bei Ausführung der Speicherbelegungsfunktion new würde der dafür vorgesehene Speicherplatz überschritten bzw. der Speicherplatz reicht nicht für den Prozeß oder die Programmeinheit aus
select_error	–	In einer select Anweisung sind alle Alternativen geschlossen und es existiert kein else-Teil
tasking_error	–	Wird im rufenden Prozeß ausgelöst, falls der gerufene Prozeß vor Annahme des Eingangsaufrufs terminiert oder bereits zur Zeit des Eingangsaufrufs terminiert hat

Hinweise:

Die Pragmas zur Unterdrückung interner Tests sind lediglich als Empfehlung an den Compiler anzusehen und schließen nicht aus, daß dennoch eine entsprechende Ausnahme ausgelöst wird (z.B. durch Verbreitung aus einer gerufenen Einheit). Zudem ist die Unterdrückung solcher Tests erst dann anzuraten, wenn ein Programm ausreichend ausgetestet ist und zuverlässig arbeitet.

Weitere Ausnahmen sind bei Benutzung der Standardpakete für die Ein- bzw. Ausgabe vordefiniert:

status_error
name_error
device_error
use_error
data_error
end_error
layout_error

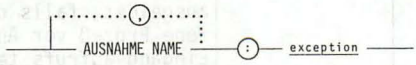
Ihre Behandlung wird ausführlich in 11.1.3 besprochen.

7.2 DIE VEREINBARUNG VON AUSNAHMEN

Syntax:

Aus

ADA 18: VEREINBARUNG



Semantik:

Der Name für eine bestimmte Ausnahmesituation wird eingeführt (etwa Singularität einer Matrix). Diese Ausnahme kann nun ausgelöst werden (7.3). Im Ausnahmebehandlungsteil (7.4) der vorliegenden (vereinbarenden) Programmeinheit oder des vorliegenden Blockes sollte eine Behandlung für diese Ausnahme angeführt sein.

Beispiele:

singulaer : exception;
iteration_divergent : exception;
ueberlauf, unterlauf : exception;

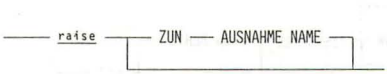
7.3 DAS AUSLÖSEN EINER AUSNAHME

Die Auslösung einer Ausnahme und der damit verbundenen Ausnahmebehandlung erfolgt explizit durch die Ausnahmeauslösung.

Syntax:

Aus

ADA 23: ANWEISUNG (ANW)



Semantik:

Es wird die Identität der Ausnahme, falls ein Name angegeben ist, festgestellt und diese Ausnahme ausgelöst. Die Ausführung des Programmes wird dann mit dem zu dieser Ausnahme gehörigen Anweisungsteil im Ausnahmebehandlungsteil fortgesetzt.

Eine Ausnahme ohne Ausnahmename kann nur in einem Ausnahme-

behandlungsteil (jedoch nicht in einem geschachtelten Unterprogramm, Paket oder Prozeß) auftreten. Sie löst erneut dieselbe Ausnahme aus, welche die Auslösung des Ausnahmebehandlungsteils bewirkte.

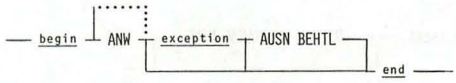
Beispiele:

```
raise singulaer;
raise;
raise iteration_divergent;
raise pool(k)'failure;
raise constraint_error;    -- explizite ausloesung
                           -- einer vordefinierten ausnahme
```

7.4 DER AUSNAHMEBEHANDLUNGSTEIL

Im Ausnahmebehandlungsteil werden die Aktionen festgelegt, die beim Auftreten einer entsprechenden Ausnahme während der Ausführung des Programmes erfolgen sollen.

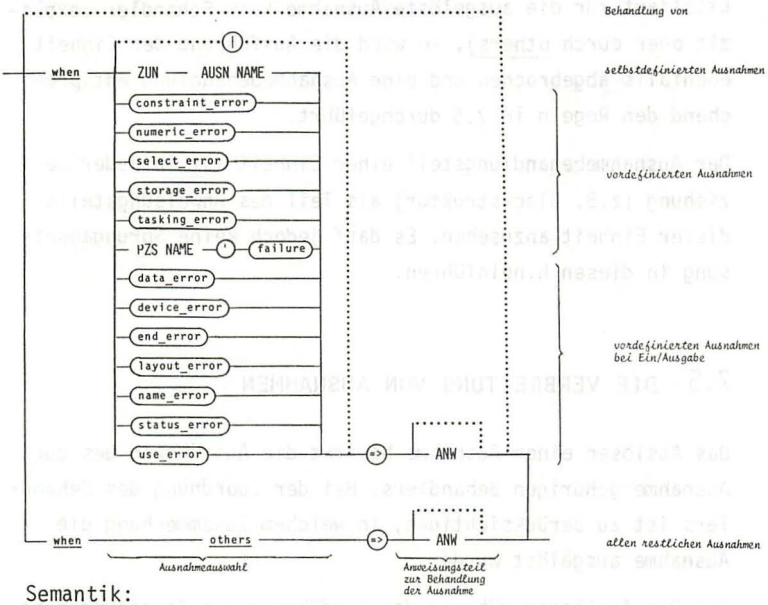
Ein Ausnahmebehandlungsteil kann im Anweisungsteil



eines Unterprogrammes (ADA 6), eines Paketrumpfes (ADA 9), eines Prozeßrumpfes (ADA 14) oder eines Blockes (ADA 24) (kurz E i n h e i t genannt) auftreten.

Syntax:

ADA 49: AUSNAHMEBEHANDLUNGSTEIL (AUSN BEHTL)



Semantik:

Der Ausnahmebehandlungsteil einer Einheit behandelt in der Regel die mit Namen versehenen Ausnahmen, welche im Anweisungsteil der Einheit ausgelöst werden. Jede Ausnahme der Einheit darf höchstens einmal als Ausnahmeauswahl im Ausnahmebehandlungsteil auftreten. Tritt in der Ausnahmeauswahl <u>others</u> auf, so werden hierunter alle vorher nicht namentlich genannten Ausnahmen behandelt einschließlich der Ausnahmen, die in der Einheit nicht sichtbar sind.

Durch eine Ausnahmeauslösung bei Ausführung des Anweisungsteiles der Einheit wird die Ausführung des zugehörigen A u s n a h m e b e h a n d l e r s , d.h. der auf die *Ausnahmeauswahl* folgenden Anweisungsfolge bewirkt. Die natürliche Ausführung des Anweisungsteils der Einheit wird dabei abgebrochen und die Ausführung der Einheit nach Ausführung

des Ausnahmebehandlers beendet. Insbesondere darf keine Sprunganweisung aus dem Ausnahmebehandlungsteil herausführen. Existiert für die ausgelöste Ausnahme kein Behandler (explizit oder durch others), so wird die Ausführung der Einheit ebenfalls abgebrochen und eine Ausnahmebehandlung entsprechend den Regeln in 7.5 durchgeführt.

Der Ausnahmebehandlungsteil einer Einheit ist in jeder Beziehung (z.B. Blockstruktur) als Teil des Anweisungsteils dieser Einheit anzusehen. Es darf jedoch keine Sprunganweisung in diesen hineinführen.

7.5 DIE VERBREITUNG VON AUSNAHMEN

Das Auslösen einer Ausnahme bewirkt die Ausführung des zur Ausnahme gehörigen Behandlers. Bei der Zuordnung des Behandlers ist zu berücksichtigen, in welchem Zusammenhang die Ausnahme ausgelöst wurde:

(a) Die Auslösung während der Ausführung von Anweisungen bewirkt die Ausführung des zugehörigen Behandlers (gekennzeichnet durch den Ausnahmenamen oder others) im selben Anweisungsteil.

Das Auslösen einer Ausnahme innerhalb eines Anweisungsteils, der keinen Ausnahmebehandler für diese Ausnahme enthält, bewirkt eine V e r b r e i t u n g der Ausnahme gemäß den folgenden Regeln:

Ausnahme ausgelöst im Anweisungsteil	bewirkt Abbruch der Ausführung bzw. Abarbeitung	und Auslösung derselben Ausnahme
eines Unterprogrammrumpfes	des Unterprogrammes	in der rufenden Programmeinheit an der Stelle des Unterprogrammaufrufs
speziell eines Hauptprogrammes	des Hauptprogrammes	—
eines Blockes	des Blockes	im Anweisungsteil, der den Block enthält
eines Paketrumpfes (Bibliothekseinheit)	des Hauptprogrammes	—
eines Paketrumpfes (keine Bibliothekseinheit)	des Paketrumpfes	in der Einheit, deren Vereinbarungsteil den Paketrumpf (bzw. den Rumpfstumpf im Fall einer Untereinheit) enthält
eines Prozeßrumpfes	des Prozesses und dessen Terminierung	—
eines Ausnahmebehandlers (jedoch nicht in einem untergeordneten Block)	der gegenwärtigen Einheit	entsprechend obigen Regeln, falls die gegenwärtige Einheit einen Unterprogrammrumpf, einen Block oder einen Paketrumpf darstellt.

Beispiel:

```
procedure p is
   error : exception;
   procedure r;
   procedure q is
   begin
      r;
      ...                    -- Nach Auslösen von error wird A
                             -- durchgeführt
   exception
      ...
      when error =>          -- Ausnahmebehandler A
      ...
   end q;
   procedure r is
   begin
      ...                    -- Nach Auslösen von error wird
                             -- die Ausnahme nach q verbreitet
                             -- und dann A durchgeführt
   end r;
begin
   ...                       -- Nach Auslösen von error wird B
   q;                        -- durchgeführt
   ...
exception
   ...
   when error =>             -- Ausnahmebehandler B
   ...
end p;
```

(b) Bei Auslösung einer Ausnahme während der Abarbeitung des Vereinbarungsteiles eines Blockes oder Rumpfes oder der Abarbeitung einer Unterprogramm-, Paket- oder Prozeßvereinbarung wird diese Abarbeitung abgebrochen und die Ausnahme auf die Einheit verbreitet, welche die Abarbeitung bewirkt hat:

Ausnahme ausgelöst bei Abarbeitung	bewirkt Abbruch der Abarbeitung bzw. Ausführung	und Auslösung derselben Ausnahme
einer Unterprogramm-, Paket-, Prozeßspezifikation (keine Bibliothekseinheit)	der Unterprogramm-, Paket-, Prozeßspezifikation	in der Einheit, welche die Spezifikation enthält
einer Unterprogramm-, Paket-, Prozeßspezifikation (keine Bibliothekseinheit)	des Programmes	—
des Vereinbarungsteils eines Unterprogrammrumpfes (kein Hauptprogramm)	des Unterprogrammes	in der rufenden Programmeinheit
des Vereinbarungsteils eines Hauptprogrammrumpfes	des Programmes	—
des Vereinbarungsteils eines Blockes	des Blockes	in der Einheit, deren Anweisungen den Block enthalten

des Vereinbarungsteils eines Paketrumpfes (keine Bibliotheksein-heit)	des Paketes	in der Einheit, welche den Paketrumpf (bzw. den Rumpfstumpf im Fall einer Untereinheit) enthält
des Vereinbarungsteils eines Paketrumpfes (Bibliothekseinheit)	des Programmes	—
des Vereinbarungsteils eines Prozeßrumpfes	des Prozesses	in der aktivierenden Einheit

Beispiel:

```
block1: declare
...
begin
   block2: declare
      n : integer := f();
   begin
   ...
   exception
      when error => ...
   end block2;
   ...
exception
   when error => ...
end block1;
```

Wird die Ausnahme error während der Initialisierung von n durch Ausführung der Funktion f ausgelöst, so wird sie im Block block1 behandelt.

(c) Spezielle Regelungen werden notwendig für die Ausnahmeverbreitung bei der Prozeßkommunikation. So können z.B. durch die nicht vorhergesehene Terminierung kommunizierender Prozesse Fehlersituationen auftreten. Eine typische Fehlersituation vor einem beabsichtigten Rendezvous liegt dann vor, wenn ein Prozeß P einen Eingang eines anderen Prozesses Q aufruft und Q ist schon terminiert. Diese Situation löst im rufenden Prozeß automatisch die vordefinierte Ausnahme tasking_error aus.

Weitere Fehlersituationen sind während eines Rendezvous' möglich:

- Tritt bei Ausführung des Anweisungsteiles einer accept-Anweisung eine Ausnahme auf, die nicht lokal behandelt wird, so wird das Rendezvous abgebrochen und die Ausnahme an den rufenden Prozeß (an die Stelle des Eingangsaufrufs) sowie an den gerufenen Prozeß verbreitet.
- Terminiert der gerufene Prozeß abnormal (etwa wegen einer abort-Anweisung), so wird die Ausnahme tasking_error an der Stelle des Eingangsaufrufs im rufenden Prozeß ausgelöst.
- Umgekehrt löst die abnormale Terminierung eines rufenden Prozesses keine Ausnahme im gerufenen Prozeß aus. Ist das Rendezvous bereits begonnen, so wird es beendet und der gerufene Prozeß bleibt unberührt; andernfalls wird der Eingangsaufruf gelöscht.
- Jeder Prozeß Q kann in einem anderen Prozeß P durch die Anweisung

  raise p'failure

  eine Ausnahme auslösen. Der die Ausnahme auslösende Prozeß Q bleibt unberührt, während für Prozeß P die Ausnahme failure eintritt. Wird P gerade ausgeführt,

so tritt p'failure an den derzeitigen Punkt der Ausführung und wird nach den üblichen Regeln behandelt. Wartet P gerade, wird die Wartezeit sofort beendet und die Ausnahme behandelt. Hat P einen Eingang gerufen, so wird dieser Aufruf gelöscht, falls das Rendezvous noch nicht begonnen hat; anderenfalls wird zunächst das Rendezvous beendet (der gerufene Prozeß bleibt unberührt).

Ein Ausnahmebehandler für die Ausnahme failure kann nur innerhalb des Prozesses P stehen.

7.6 BEISPIEL

Im folgenden Programm, welches 20 2×2-Matrizen einliest, invertiert und ausgibt, wird eine Ausnahme singulaer für den Fall behandelt, daß eine zu invertierende Matrix singulär ist. Bei Auslösung der Ausnahme singulaer findet eine Verbreitung in die Prozedur p statt. Dort wird ein entsprechender Kommentar ausgegeben und die Ausführung von p beendet. Danach wird die Ausführung des Programmes in der for-Schleife der Prozedur matinv mit dem nächsten Wert i fortgesetzt.

Zur Ein- und Ausgabe von Matrizen wird das Paket mat2_2_ea verwendet, das in 11.2.2.4 vereinbart wird.

```
with mat2_2_ea; use mat2_2_ea;
with text_io;   use text_io;
procedure matinv is
   type matrix is array (1 .. 2,1 .. 2) of float;
  singulaer : exception;  -- Ausnahme singulaer verein-
                          -- bart
```

```
procedure invert (m : in out matrix) is
   det,a,b : float;
   epsilon : constant float := 1.Ø e-1Ø;
   begin
      det:= m(1,1)*m(2,2)-m(1,2)*m(2,1);
      if abs(det) < epsilon then
         raise singulaer; end if;  -- Ausloesung von
                                   -- singulaer
      b:= 1.Ø/det; a:= m(1,1); m(1,1):= m(2,2)*b;
      m(2,2):= a*b; m(1,2):= -m(1,2)*b;
      m(2,1):= -m(2,1)*b;
   end;
procedure p is
   x : matrix;
   begin
      read(x); invert(x); print(x);
      exception
      when singulaer => put("Matrix ist singulaer");
      when others => put("unerwarteter Fehler");
   end;
begin
   for i in 1 .. 2Ø loop
      put("Iteration"); put(i);
      p;
   end loop;
end;
```

8. COMPILERINSTRUKTIONEN

Während der Übersetzung eines Programmes treten Fragen auf, die nichts mit der Logik des Programmes zu tun haben, deren Beantwortung jedoch für den erfahrenen Programmierer interessant sein kann. Bei Kenntnis der Zielmaschine kann man zum Beispiel durch genaue Festlegung der Speicherplatzaufteilung für eine Variable eine Beschleunigung der Adressierung erzielen. Wenn man von der Richtigkeit eines Programmes überzeugt ist, kann man etwa auf eine Indexüberprüfung verzichten, um die Laufzeit des Programmes zu beschleunigen.

8.1 DIREKTIVEN (PRAGMAS)

Mit Hilfe einer D i r e k t i v e (eines P r a g m a s) werden dem Compiler gewisse Informationen über den Verlauf der Übersetzung gegeben. Zu den hier vorgestellten sprachabhängigen Direktiven können noch weitere implementierungsabhängige definiert werden. Eine Direktive, deren Namen der Compiler nicht kennt, hat keinen Einfluß auf die Wirkung des Programmes.

Der syntaktische Aufbau aller Pragmas ist

— pragma — NAME — AKT DIR ARG LISTE —

der Direktive *Aktuelle Argumentliste für diese Direktive*

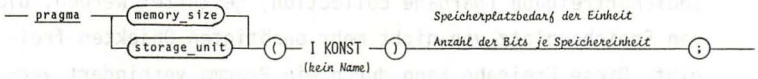
Im implementierungsabhängigen Paket system sind unter anderem Konstanten zusammengefaßt, die die Speicherplatzaufteilung festlegen. Diese können umdefiniert werden.

Speicherplatzaufteilung:

Syntax:

Aus

ADA 52: PRAGMA



Semantik:

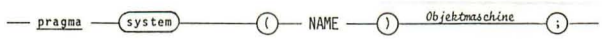
Diese Pragmas stehen vor einer Bibliothekseinheit. memory_size legt den Speicherplatzbedarf der Bibliothekseinheit in Speicherplatzeinheiten fest. storage_unit bestimmt die Anzahl der Bits pro Speichereinheit für diese Bibliothekseinheit. Beide überschreiben die im Paket system stehenden I Konstanten gleichen Namens.

Festlegung der Objektmaschine:

Syntax:

Aus

ADA 52: PRAGMA



Semantik:

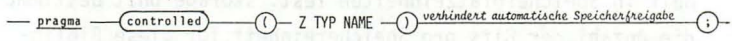
Dieses Pragma steht vor einer Bibliothekseinheit. Es legt die Objektmaschine fest, mit der diese Bibliothekseinheit ausgeführt werden soll. Die AZ Konstante name des speziellen AZ Typs system.system_name wird dadurch überschrieben.

Speicherverwaltung:

Objekte vom referierten Typ eines Zeigertyps werden während der Laufzeit generiert und in den Speicher gelegt. Ist der vorhandene Speicherplatz belegt, so kann eine automatische Speicherfreigabe (garbage collection) gestartet werden, die den Speicherplatz von nicht mehr benötigten Objekten freigibt. Diese Freigabe kann durch ein Pragma verhindert werden.

Syntax:

ADA 52: PRAGMA



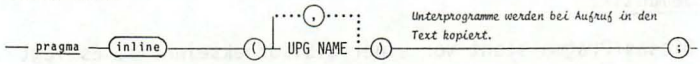
Semantik:

Die automatische Speicherfreigabe für Objekte vom referierten Typ wird solange verhindert, bis die Gültigkeit des Zeigertyps erlischt. Das Pragma steht im Vereinbarungsteil der Programmeinheit, in der der Zeigertyp vereinbart wurde, dort, wo auch eine Darstellungsangabe stehen kann.

Aufruf von Unterprogrammen:

Syntax:

ADA 52: PRAGMA



Semantik:

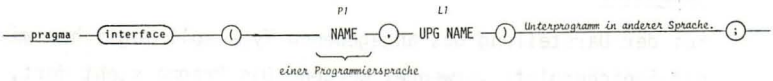
Die Rümpfe der aufgeführten Unterprogramme werden nach Ersetzung der formalen durch die aktuellen Argumente an der

Stelle ihres Aufrufes in den erzeugten Code kopiert. Ist der Unterprogrammname überladen, so wirkt das Pragma für alle Versionen. Das Pragma steht im gleichen Vereinbarungsteil wie die Unterprogramme.

Die Wirkung eines Unterprogrammes wird durch das Pragma inline nicht verändert.

Syntax:

Aus ADA 52: PRAGMA



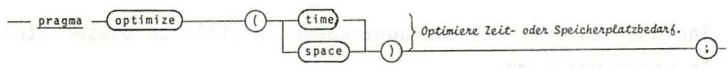
Semantik:

Das Unterprogramm P1 ist in der angegebenen Programmiersprache L1 geschrieben. Das Pragma steht nach der ADA-Spezifikation dieses Unterprogrammes im gleichen Vereinbarungsteil. Für das Unterprogramm darf kein Rumpf oder Rumpfstumpf folgen. Der Code des von einem L1-Compiler übersetzten Unterprogrammes P1 steht dem ADA-Compiler zur Verfügung. Die Aufrufkonventionen der Sprache L1 sind zu beachten.

Optimierung:

Syntax:

Aus ADA 52: PRAGMA



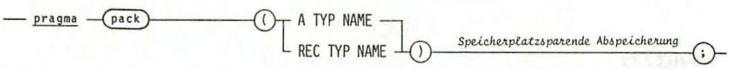
Semantik:

Optimiere die Laufzeit bzw. den Speicherplatzbedarf der Pro-

grammeinheit, in dessen Vereinbarungsteil das Pragma auftritt.

Syntax:

Aus ADA 52: PRAGMA



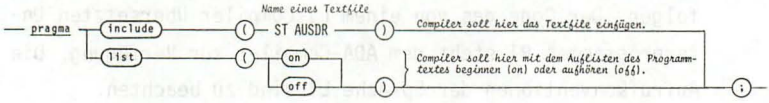
Semantik:

Bei der Darstellung des angegebenen Typs soll möglichst wenig Speicherplatz verwendet werden. Das Pragma steht dort, wo eine Darstellungsangabe für den Typ stehen kann.

Programmtext:

Syntax:

Aus ADA 52: PRAGMA



Semantik:

Diese Pragmas können überall dort im Programm eingefügt werden, wo ein Semikolon steht, und vor der ersten Programmzeile.

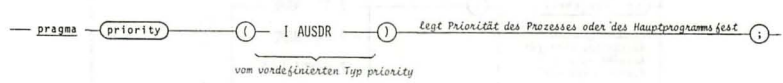
include bewirkt, daß das angegebene Textfile an dieser Stelle eingefügt wird.

list on bedeutet, daß ab dieser Stelle der Programmtext vom Compiler aufgelistet werden soll, list off bedeutet, daß die Auflistung des Programmtextes beendet werden soll.

Priorität von Prozessen:

Syntax:

Aus ADA 52: PRAGMA



Semantik:

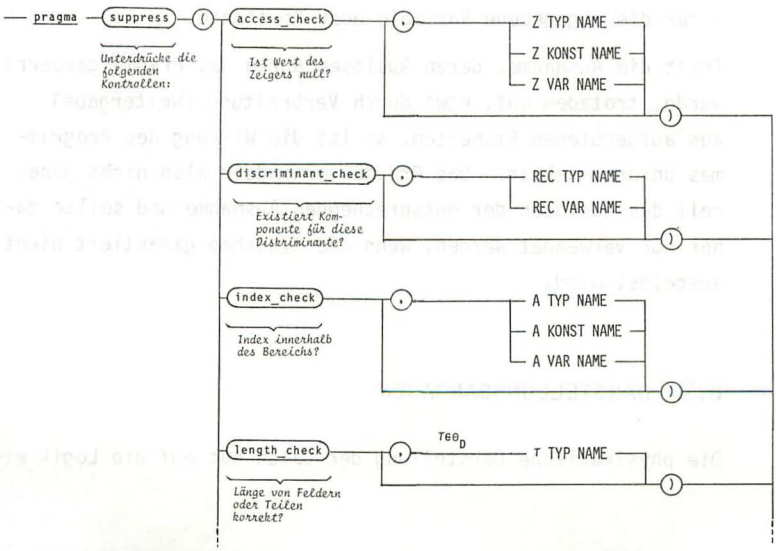
Der statische Ausdruck vom Typ priority wird ausgewertet und dem Prozeß oder dem Hauptprogramm als Priorität zugewiesen (siehe 9.6).

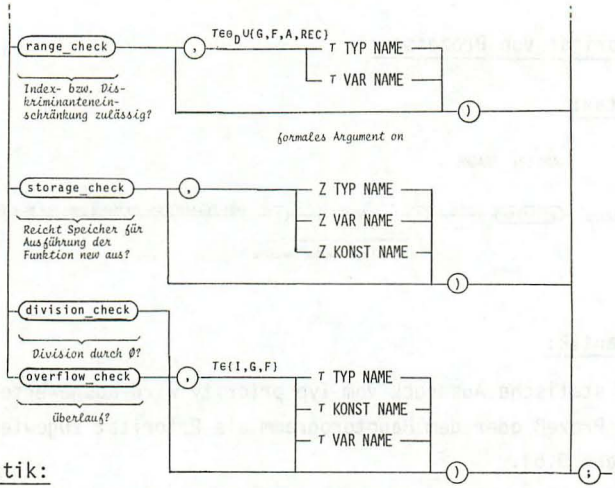
Das Pragma darf nur im äußersten Vereinbarungsteil des Hauptprogrammes oder in einer Prozeßspezifikation stehen.

Unterdrücken von Tests:

Syntax:

Aus ADA 52: PRAGMA





Semantik:

Das Pragma steht im Vereinbarungsteil einer Einheit. Es bewirkt, daß in dieser Programmeinheit der angegebene Test (siehe 7.1) ausgeschaltet wird, und zwar

- für alle Elemente der in Frage kommenden Typen, falls kein Argument angegeben wurde
- für alle Elemente eines Typs, falls ein Typname als Argument angegeben wurde
- für die angegebene Variable oder Konstante.

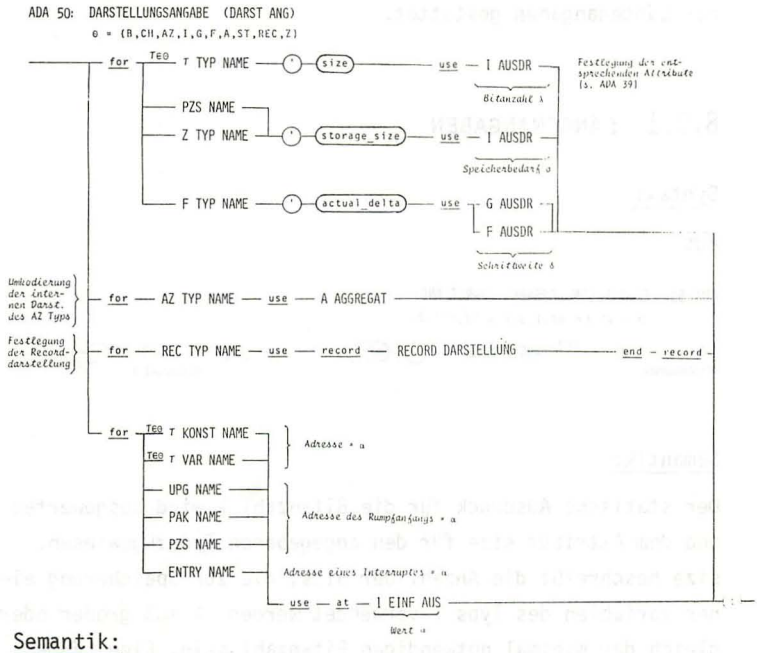
Tritt die Ausnahme, deren Auslösen durch das Pragma gesperrt wurde, trotzdem auf, etwa durch Verbreitung (Weitergabe) aus aufgerufenen Einheiten, so ist die Wirkung des Programmes unvorhersehbar. Das Pragma verhindert also nicht generell das Auslösen der entsprechenden Ausnahme und sollte daher nur verwendet werden, wenn die Ausnahme garantiert nicht ausgelöst wird.

8.2 DARSTELLUNGSANGABEN

Die physikalische Darstellung der Daten hat auf die Logik ei-

nes Programmes keine Wirkung, daher reicht im Normalfall eine vom Compiler gewählte Darstellung aus. (Unter der D a r - s t e l l u n g einer Größe verstehen wir etwa die Speicherplatzaufteilung, die Adresse, die Reihenfolge der Komponenten einer Record Variablen, die internen Werte für die Konstanten eines Aufzählungstyps). Zur Erhöhung der Effizienz oder zur Kommunikation mit anderen Programmen oder Geräten kann es jedoch wichtig sein, die genaue Darstellung einer Größe zu bestimmen.

Syntax:



Semantik:

Eine Darstellungsangabe legt gewisse Details der physikalischen Darstellung einer Größe fest und überschreibt die normalerweise vom Compiler gewählte Darstellung. Die Wirkung der Darstellungsangaben wird im einzelnen später beschrie-

ben.

Eine Darstellungsangabe wirkt auf solche Größen, die im gleichen Vereinbarungsteil vereinbart wurden. Für einen Typ können mehrere Darstellungsangaben nur dann gegeben werden, wenn sie verschiedene Eigenschaften beschreiben. Die Darstellungsangabe gilt für alle Größen eines Typs; will man unterschiedliche Darstellungen für verschiedene Variable desselben Typs, so muß man einen abgeleiteten Typ definieren. Es gibt keine Darstellungsangaben für abgeleitete Typen von Zeigertypen. Für abgeleitete Typen, die auch benutzerdefinierte Unterprogramme von ihren Basistypen übernehmen, sind nur Längenangaben gestattet.

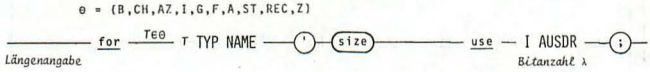
8.2.1 LÄNGENANGABEN

Syntax:

Aus

ADA 50: DARSTELLUNGSANGABE (DARST ANG)

θ = {B,CH,AZ,I,G,F,A,ST,REC,Z}



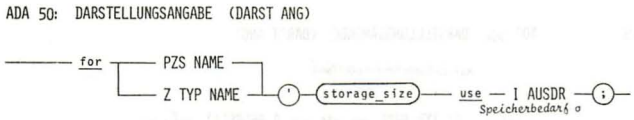
Semantik:

Der statische Ausdruck für die Bitanzahl λ wird ausgewertet und dem Attribut size für den angegebenen Typ zugewiesen. size beschreibt die Anzahl der Bits, die zur Speicherung einer Variablen des Typs T verwendet werden. λ muß größer oder gleich der minimal notwendigen Bitanzahl sein. Eine Längenangabe für einen strukturierten Typ wirkt nicht auf die Komponenten, sondern höchstens auf die Lücken zwischen den Komponenten. Eine Längenangabe für einen Typ mit dynamischen Eigenschaften, etwa unbeschränkter Indexbereich, ist nicht

erlaubt.

Syntax:

Aus

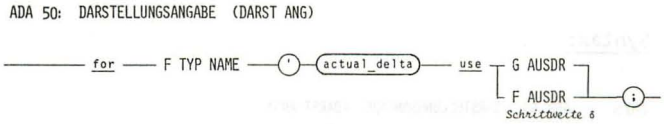


Semantik:

Der I Ausdruck für den Speicherbedarf σ wird ausgewertet und dem Attribut storage_size für den angegebenen Z Typ oder Prozeß zugewiesen. storage_size beschreibt für einen Prozeß oder Prozeßtyp die Anzahl der Speichereinheiten, die zur Aktivierung des Prozesses verwendet werden, für einen Zeigertyp die Anzahl der Speichereinheiten, die für alle im Verlauf des Programmes erzeugten Objekte des referierten Typs dieses Zeigertyps bereitgestellt werden.

Syntax:

Aus



Semantik:

Der reelle Ausdruck δ, der nur Literalkonstanten beinhalten darf, wird ausgewertet und dem Attribut actual_delta für den Festpunkttyp zugewiesen. actual_delta beschreibt die tatsächlich für die Darstellung des Festpunkttyps verwendete Schrittweite. δ muß kleiner oder gleich der in der Typvereinbarung gegebenen Schrittweite sein.

8.2.2 AUFZÄHLUNGSTYPDARSTELLUNG

Syntax:

Aus ADA 50: DARSTELLUNGSANGABE (DARST ANG)

Aufzählungstypdarstellung

— for — AZ TYP NAME — use — A AGGREGAT —(;)—

Semantik:

Durch das Aggregat, dessen Komponententyp integer und dessen Indextyp der angegebene AZ Typ ist, werden die internen Werte für die AZ Konstanten festgelegt. Die integer Werte müssen der Ordnung des AZ Typs entsprechen. Die attributierten Funktionen succ, pred und pos werden durch eine Darstellungsangabe in ihrer Wirkung nicht beeinflußt, sie gelten für den Typ, wie er vereinbart wurde.

8.2.3 RECORDDARSTELLUNG

Syntax:

Aus ADA 50: DARSTELLUNGSANGABE (DARST ANG)

— for — REC TYP NAME — use — record — RECORD DARSTELLUNG ———— end — record —(;)—

ADA 51: RECORD DARSTELLUNG

— at — mod — I EINF AUS —(;)— (Wert v) | T KOMP NAME — at — I EINF AUS — range — I EINF AUS —(..)— I EINF AUS —(;)—

Position u first_bit λ last_bit ρ

Semantik:

Alle integer Ausdrücke ν, μ, λ und ρ sind statisch. Die Anfangsadresse des Records muß ein Vielfaches von ν sein.

Für jede angegebene Komponente werden die Attribute position auf μ, first_bit auf λ und last_bit auf ρ gesetzt.

Innerhalb eines Records werden die relativen Positionen μ der einzelnen Komponenten in Speichereinheiten angegeben. Die Numerierung beginnt bei 0. Die Positionen der tatsächlich belegten Bits werden relativ zum Beginn der Komponente bei 0 durch λ (erstes Bit) und ρ (letztes Bit) beschrieben.

Der angegebene Speicherbereich muß ausreichend groß sein.

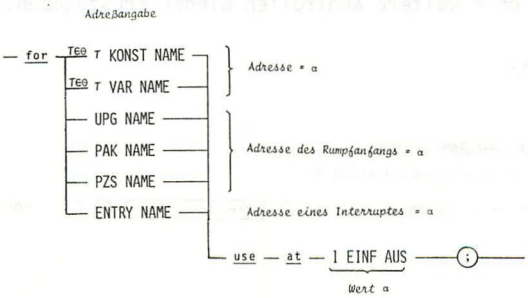
Nur für Komponenten oder Diskriminanten, die eine statische Einschränkung haben, dürfen Darstellungen angegeben werden. Darstellungsangaben für Komponenten mit dynamischen Eigenschaften sind nicht erlaubt.

8.2.4 ADRESSANGABEN

Syntax:

Aus

ADA 50: DARSTELLUNGSANGABE (DARST ANG)



Semantik:

Der statische Ausdruck α bestimmt

- bei Variablen und Konstanten die Adresse
- bei Unterprogrammen, Paketen oder Prozessen die Anfangsadresse des Maschinencodes des Rumpfes
- bei Eingängen eine Adresse, die einen Hardware-Interrupt beschreibt.

Durch solche Adreßangaben dürfen keine Überlagerungen des Speichers (Overlays) hervorgerufen werden.

Eine Hardware-Unterbrechung (Interrupt) darf nur mit höchstens einem Eingang verbunden werden. Das Auftreten einer Unterbrechung wirkt wie ein Eingangsaufruf von einem Prozeß, dessen Priorität höher ist als die aller Benutzerprozesse (siehe 9.6). Die Behandlung von Unterbrechungen, z.B. das "Übertragung beendet" - Signal eines Druckers, kann so in ADA programmiert werden.

8.3 UNKONTROLLIERTE SPEICHERFREIGABE

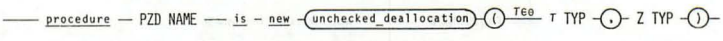
Eine Ausprägung des Prozedurschemas unchecked_deallocation bietet die Möglichkeit, den durch Generierung einer Variablen vom referierten Typ eines Zeigertyps belegten Speicherplatz ohne weitere Kontrollen wieder freizugeben.

Syntax:

Aus

ADA 8: UNTERPROGRAMM ERZEUGUNG

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z}



referierter Typ T

Semantik:

Es wird eine Prozedur p erzeugt, die eine Zeigervariable x als formales Ein/Ausgabeargument hat. Ein Aufruf der Prozedur p bewirkt, daß der Wert des aktuellen Argumentes x:= null gesetzt wird. Die von x referierte Variable wird nicht mehr benötigt, und ihr Speicherplatz kann freigegeben werden. Falls eine weitere Zeigervariable auf die gleiche Variable verwies, so darf dieser Verweis jetzt nicht mehr benutzt werden.

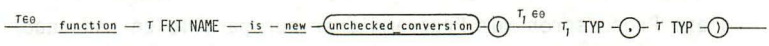
8.4 UNKONTROLLIERTE TYPUMWANDLUNG

Der Wert eines jeden Typs wird durch ein Bitmuster dargestellt. Das gleiche Bitmuster hat für verschiedene Typen auch verschiedene Bedeutung. Es kann für den Programmierer interessant sein, ein und dasselbe Bitmuster als Wert für verschiedene Typen zu interpretieren.

Syntax:

Aus

ADA 8: UNTERPROGRAMM ERZEUGUNG
Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z}



Semantik:

Es wird eine Funktion f erzeugt, die ein formales Eingabeargument x vom Typ T_1 hat und deren Ergebnistyp T ist. T_1 und T haben die gleiche Darstellung. Durch einen Aufruf von f wird das Bitmuster, das das aktuelle Argument x repräsentiert, als Wert des Typs T interpretiert. Tatsächlich findet also keine Aktion statt, das Bitmuster wird nur anders interpretiert. Das Funktionsergebnis muß stets ein definierter Wert vom Typ T sein.

9. HANDHABUNG VON PROZESSEN

Prozesse (*tasks*) bilden neben Unterprogrammen und Paketen die dritte Form von Programmeinheiten. Sie ermöglichen die Entwicklung von Programmsystemen, deren Komponenten entweder gleichzeitig miteinander ablaufen müssen, oder zumindest so weitgehend unabhängig voneinander sind, daß bei einer entsprechenden sequentiellen Notation Programmstruktur und Lösungsidee kaum noch zu erkennen wären. Wenn hier nur weitgehende, also nicht unbedingt vollständige Unabhängigkeit der Prozesse gefordert wird, dann deshalb, weil unter dem grundlegenden Begriff des Rendezvous' Möglichkeiten der Zusammenarbeit von und des Datenaustausches zwischen parallelen Prozessen geboten werden.

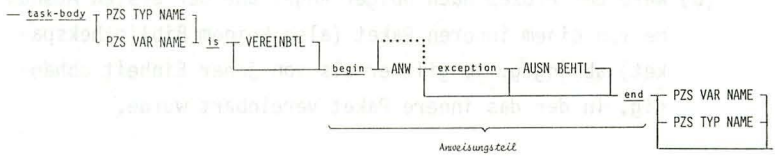
Prozesse besitzen jedoch nicht nur eine große Bedeutung für die Entwicklung und bessere Strukturierung von Programmsystemen, sondern sie bieten auch die Möglichkeit zur Erhöhung der Ausführungsgeschwindigkeit von solchen Programmsystemen. Moderne Großrechenanlagen (oder Rechnerverbunde) stellen nämlich in der Regel mehr als einen Zentralprozessor zur Verfügung. Durch die Programmierung paralleler Prozesse können diese Prozessoren vom Programmierer zur parallelen und unabhängigen Ausführung von Programmstücken veranlaßt werden, wobei die Anzahl der zur Verfügung stehenden Prozessoren dem Programmierer nicht bekannt sein muß.

Prozesse sind gemäß Diagramm ADA 1 keine selbständigen *Übersetzungseinheiten* (außer in der Form von *Untereinheiten*); sie werden als Objekte eines Prozeßtyps im Vereinbarungsteil einer umfassenden Programmeinheit (eines Blocks, eines Unterprogrammes, eines Prozesses oder eines Bibliothekpaketes) - dem sogenannten V o r g ä n g e r - vereinbart oder dynamisch durch Aufruf der Funktion new zu einem Zeiger mit

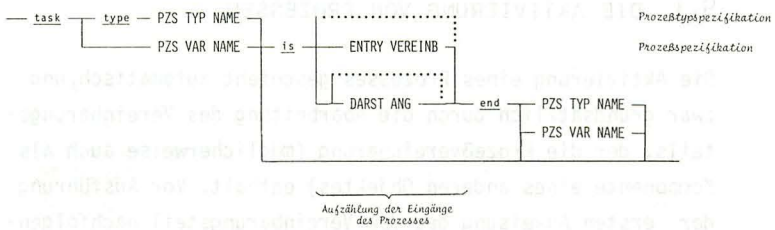
einem Prozeßtyp als referiertem Typ geschaffen. Die Vereinbarung von Prozessen wird in Abschnitt 2.6 behandelt. Wir wiederholen hier die entsprechende

Syntax:

ADA 14: PROZESSRUMPF



ADA 15: PROZESSSPEZIFIKATION



Syntaktisch ist die Prozeßvereinbarung in eine Prozeßspezifikation und den dazugehörigen Prozeßrumpf unterteilt.

Spezifikation und Rumpf eines Prozesses müssen im gleichen Vereinbarungsteil stehen. Eine Ausnahme von dieser Regel bilden Prozesse in Paketen: ihre Spezifikationen können bereits außerhalb vom Vereinbarungsteil des Paketrumpfes im sichtbaren oder geschützten Teil der Paketspezifikation stehen.

Soll der Prozeßrumpf in Form einer Untereinheit getrennt übersetzt werden, muß er im Vereinbarungsteil zunächst durch einen Stumpf ersetzt werden (ADA 17).

Jeder Prozeß ist von einer Programmeinheit a b h ä n g i g (dependent), und zwar von jener, in der der betreffende Pro-

zeß vereinbart ist, also von seinem Vorgänger. Von dieser Regel gibt es zwei Ausnahmen:

(a) Wird der Prozeß selbst oder als Komponente eines Objekts durch eine Zeigervariable referiert, so ist er von jener Einheit abhängig, in der der entsprechende Zeigertyp vereinbart wurde.

(b) Wäre der Prozeß nach obiger Regel und der ersten Ausnahme von einem inneren Paket (also keinem Bibliothekspaket) abhängig, so gilt er als von jener Einheit abhängig, in der das innere Paket vereinbart wurde.

9.1 DIE AKTIVIERUNG VON PROZESSEN

Die Aktivierung eines Prozesses geschieht automatisch, und zwar grundsätzlich durch die Abarbeitung des Vereinbarungsteils, der die Prozeßvereinbarung (möglicherweise auch als Komponente eines anderen Objektes) enthält. Vor Ausführung der ersten Anweisung des dem Vereinbarungsteil nachfolgenden Anweisungsteils sind also alle Prozesse des vorangehenden Vereinbarungsteils aktiviert und werden nun parallel und unabhängig voneinander bis zu ihrer Terminierung (siehe 9.5) ausgeführt. Ist der Anweisungsteil des Vorgängers leer (etwa bei einem Paketrumpf) oder fehlt der Rumpf total, so werden die Prozesse so aktiviert, als ob ein Rumpf mit einer leeren Anweisung vorhanden wäre.

Wird ein Prozeß dynamisch als referierte Variable oder Komponente einer referierten Variable geschaffen, so wird er automatisch aktiviert und seine Ausführung begonnen. Der entsprechende Aufruf der Funktion new ist immer erst dann vollständig ausgeführt, wenn alle dabei neu geschaffenen Prozesse aktiviert sind.

Die Ausführung eines Prozesses ist beendet, wenn das Ende

der Anweisungen des Prozeßrumpfes erreicht ist, oder wenn er sich selbst terminiert oder von einem anderen Prozeß oder dem Vorgänger terminiert wird. Die Ausführung des Vorgängers ist beendet, wenn alle in ihm aktivierten Prozesse beendet sind (siehe 9.5).

Beispiel:

```
with text_io; use text_io;
procedure parallele_prozesse is
   task drucken;
   task body drucken is
   begin
      loop put("ada ist dufte");
           new_line;
           delay 60.0;
      end loop;
   end drucken;
   task zaehlen;
   task body zaehlen is
   i : integer;
   begin
      loop i:= 0;
           loop i:= i + 1;
                exit when i = 100;
           end loop;
      end loop;
   end zaehlen;
   begin
      null;
   end;
```

Beim Eintritt in den Anweisungsteil der Prozedur sind die beiden Prozesse drucken und zaehlen aktiviert. Damit werden der Prozeduranweisungsteil und die beiden Prozesse parallel

ausgeführt. Die Ausführung der Prozedur endet nicht vor der Terminierung der beiden Prozesse, d.h. im vorliegenden Fall nie.

Hinweis :

(1) Es ist nicht festgelegt, in welcher Reihenfolge die Prozesse eines Vereinbarungsteils aktiviert werden.

9.2 DAS RENDEZVOUS VON PROZESSEN

Entsprechend seiner Aufgabenstellung wird ein Prozeß gewisse Dienstleistungen für andere Prozesse anbieten bzw. Dienstleistungen anderer Prozesse abrufen müssen; im ersten Fall beinhaltet der Prozeß passive Elemente und heißt daher passiver Prozeß oder slave-Prozeß, während er im zweiten Fall aktive Elemente enthält und daher aktiver Prozeß oder master-Prozeß heißt. Der Austausch, d.h. die Abgabe bzw. Annahme solcher Dienstleistungen zwischen zwei Prozessen geschieht während eines sogenannten R e n d e z v o u s , zu dem die beiden Prozesse synchronisiert werden und während dem der Austausch von Daten möglich wird.

Bevor wir auf Einzelheiten näher eingehen, wollen wir das Prinzip des Rendezvous' allgemeiner betrachten:

(a) Ein passiver Prozeß stellt Dienstleistungen zur Verfügung. Jede dieser Dienstleistungen kann über einen Eingang des passiven Prozesses von einem beliebigen aktiven Prozeß angesprochen werden. Ein passiver Prozeß stellt seine Dienstleistungen gewissermaßen unkontrolliert zur Verfügung. Jeder beliebige Prozeß, welcher einen entsprechenden Eingang aufruft, wird bedient.

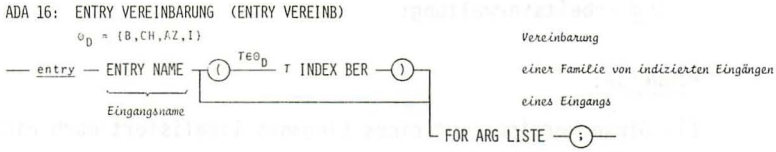
(b) Ein aktiver Prozeß hingegen muß den Namen des passiven Prozesses und den Namen des Eingangs, welchen er aufruft, wie auch die zu erwartende Dienstleistung kennen.

Treten während der Abarbeitung Fehler auf, so werden diese vom aufgerufenen Prozeß an den aufrufenden weitergegeben, aber nicht umgekehrt.

9.2.1 DIE VEREINBARUNG EINES EINGANGS

Wie schon erwähnt kann ein Prozeß Dienstleistungen zur Verfügung stellen, die über bestimmte Eingänge (entries) aufgerufen werden können. Die Eingänge müssen in der Prozeßvereinbarung in folgender Form aufgeführt werden:

Syntax:



Semantik:

Eine entry-Vereinbarung legt bei der Vereinbarung eines Prozeßtyps den Namen für einen Eingang bzw. - bei Angabe eines Indexbereiches - für eine ganze Familie (nämlich ein eindimensionales Feld) von verschiedenen Eingängen mit derselben formalen Argumentliste fest.

Daher besitzen alle Prozesse eines Prozeßtyps Eingänge mit dem gleichen Namen, die von außerhalb dieser Prozesse mit dem Prozeßnamen als Zunamen und einer passenden aktuellen Argumentliste aufgerufen werden können (siehe 9.2.2).

Eine Vereinbarung eines Eingangs entspricht völlig einer Prozedurvereinbarung. Die Verarbeitung der formalen Argumentliste verläuft entsprechend (siehe 5.1).

Beispiele:

```
1. task type roboter is
      entry taufe(wer_bin_ich : robbie_zeiger);
      entry lebenszeichen;
   end roboter;

2. task boss is
      entry empfang(kontakt)(gespraech: in geschaeft);
      entry lager_infos(nachricht: in string);
   end boss;

3. task arbeitsverwaltung is
      entry auftragsannahme(auftrag:in datenblock);
      entry arbeitsvergabe(auftrag: out datenblock);
   end arbeitsverwaltung;
```

Hinweise:

(1) Diese Vereinbarung eines Eingangs lokalisiert noch nicht die Lage des Eingangs im Prozeßrumpf. Dies geschieht erst durch eine entsprechende accept-Anweisung (siehe 9.2.3).

(2) Ein Eingang kann in eine Prozedur umbenannt werden (siehe 2.8).

(3) Die einzelnen Eingänge einer Familie werden wie die Komponenten eines eindimensionalen Feldes angesprochen.

(4) Ein Eingangsname kann überladen werden, und zwar sowohl mit weiteren Eingängen als auch mit Prozeduren.

9.2.2 DER AUFRUF EINES EINGANGS

Ein Prozeß kann die Eingänge eines anderen Prozesses aufrufen.

Syntax: Aus ADA 23: ANWEISUNG

Semantik:

Die Wirkung eines Eingangsaufrufs kann am besten im Zusammenhang mit einer entsprechenden accept-Anweisung im gerufenen Prozeß beschrieben werden (siehe 9.2.4). Für den Eingangsaufruf sind folgende Punkte zu beachten:

- Ein Eingangsaufruf darf nur in einem Prozeß oder im Vorgänger des Prozesses auftreten, der den Eingang enthält.
- Der Zuname muß immer den Namen des Prozesses enthalten, in dem der aufgerufene Eingang vereinbart wurde.
- Die aktuelle Argumentliste muß zu der formalen Argumentliste der Entryvereinbarung passen, wie bei einem Prozeduraufruf die aktuelle Argumentliste zu der formalen Argumentliste der Prozedurvereinbarung passen muß (vgl. 5.6).
- Falls es sich um einen Eingang aus einer Familie handelt, wird zuerst der Indexausdruck ausgewertet, danach die aktuellen Argumente.

Beispiele:

```
arbeitsverwaltung.arbeitsvergabe(auftrag);
boss.lager_infos("meldung");
```

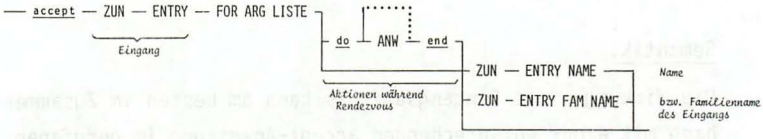
9.2.3 DIE ANNAHMEANWEISUNG (accept-ANWEISUNG)

Die Aktionen, die beim Aufruf eines Eingangs durchgeführt werden sollen, stehen in einer accept-Anweisung im Rumpf des

Prozesses, in dem der Eingang vereinbart wurde.

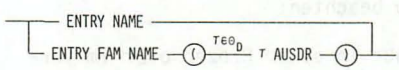
Syntax:

ADA 26: ACCEPT ANWEISUNG



ADA 64: ENTRY

$\Theta_D = \{B, CH, AZ, I\}$



Semantik:

Die Wirkung der accept-Anweisung wird im Zusammenhang mit einem entsprechenden Eingangsaufruf beschrieben (siehe 9.2.4). Es sind folgende Punkte zu beachten:

(a) Eine accept-Anweisung darf nur im Rumpf des Prozesses stehen, in dem der Eingang vereinbart wurde.

(b) Für einen Eingang können mehrere accept-Anweisungen vorhanden sein.

(c) Die formale Argumentliste in der ACCEPT ANWEISUNG muß mit der in der zugehörigen ENTRY VEREINBARUNG übereinstimmen. Es gelten die gleichen Regeln wie bei einer Prozedur und einer separaten Prozedurspezifikation.

Hinweise:

(1) Jeder Prozeß kann nur accept-Anweisungen für seine eigenen Eingänge enthalten.

(2) Der Rumpf einer accept-Anweisung kann weitere accept-Anweisungen oder Eingangsaufrufe enthalten.

Beispiele:

1. accept taufe(wer_bin_ich: robbie_zeiger) do
   eigene_person:= wer_bin_ich;
   end taufe;

2. accept lager_infos(nachricht : in string);

3. accept arbeitsvergabe(auftrag : out datenblock) do
   auftrag:= aktueller_auftrag;
   end arbeitsvergabe;

9.2.4 DIE SYNCHRONISIERUNG

Die Synchronisierung von Prozessen erfolgt durch das Zusammenspiel von Eingangsaufruf und accept-Anweisung in zwei verschiedenen Prozessen. Sie erfordert also grundsätzlich einen aktiven und einen passiven Prozeß. Der Ablauf der Synchronisierung geschieht in folgender Form:

Die Ausführung eines Eingangsaufrufes in einem aktiven (rufenden) Prozeß hat zunächst die Wirkung, daß die Ausführung des Prozeßrumpfes ruht, bis in dem durch Eingangsaufruf bezeichneten passiven (gerufenen) Prozeß eine accept-Anweisung für den gerufenen Eingang zur Ausführung gelangt.

Gelangt umgekehrt ein passiver Prozeß an eine accept-Anweisung für einen Eingang, zu welchem noch kein Eingangsaufruf vorliegt, so wird die Ausführung des Prozedurrumpfes unterbrochen bis ein entsprechender Eingangsaufruf vorliegt.

Dann erfolgt die Ersetzung der formalen durch die aktuellen Argumente wie bei einem Prozeduraufruf, und der Rumpf der accept-Anweisung wird ausgeführt. Während dieser Zeit bleiben die beiden Prozesse zusammen und führen keine selbständigen Aktionen durch. Diesen Vorgang nennt man R e n d e z v o u s .

Nach Ausführung des Rumpfes der accept-Anweisung werden die Argumente wieder analog zum Prozeduraufruf behandelt. Anschließend setzen die beiden Prozesse ihre Verarbeitung mit der auf den Eingangsaufruf bzw. die accept-Anweisung folgenden Anweisung wieder parallel und asynchron fort.

Rufen mehrere Prozesse den gleichen Eingang eines passiven Prozesses auf, so werden sie in eine W a r t e s c h l a n g e für diesen Eingang eingereiht. Beim Erreichen einer accept-Anweisung wird jeweils der erste Prozeß der Warteschlange bedient.

Die Einreihung in die Warteschlange erfolgt nach dem Prinzip "Wer zuerst kommt, mahlt zuerst" (first-in, first-out).

Hinweise:

(1) Man sollte im Rumpf einer accept-Anweisung nur diejenigen Aktionen beschreiben, die für die Synchronisation und den Datenaustausch zwischen den Prozessen unbedingt nötig sind. Denn zur Zeit des Rendezvous' findet ja keine Parallelarbeit statt.

(2) Ein Eingang kann aufgerufen werden, bevor der entsprechende Prozeß aktiviert wurde.

(3) Ein Prozeß kann seine eigenen Eingänge aufrufen, doch führt dies in eine Sackgasse.

(4) Es kann nicht vorkommen, daß ein Prozeß in mehreren Warteschlangen steht.

9.3 DAS BEDINGTE RENDEZVOUS

Das bedingungslose Warten auf ein Rendezvous, wie es in der bisher besprochenen Form mit entry-Aufrufen und accept-Anweisungen programmiert wird, kann zu unerwünschten Situatio-

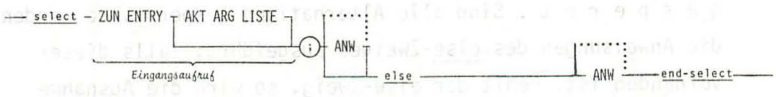
nen führen. Würde etwa ein Prozeß einen von mehreren Eingängen eines anderen Prozesses aufrufen, dieser jedoch zuerst an eine accept-Anweisung für einen anderen als den aufgerufenen Eingang gelangen, wären beide Prozesse - evtl. für immer - blockiert. Man denke z.B. daran, daß es sicher sinnlos ist, etwas auf einem Drucker ausgeben zu wollen, falls keiner am System angeschlossen ist, oder daß ein Bankkassierer unbedingt darauf wartet, einem Kunden etwas auszuzahlen, obwohl nur Kunden warten, die etwas einzahlen wollen.

Daher ist eine - laufzeitabhängige - Auswahl zwischen alternativen Anweisungen notwendig.

Es gibt in ADA verschiedene Möglichkeiten, eine flexible Kommunikation zwischen Prozessen zu ermöglichen. Die einfachste ist das b e d i n g t e Rendezvous, bei welchem der aufrufende Prozeß einen bedingten Eingangsaufruf durchführt.

Syntax: Aus

ADA 24: STRUKTURIERTE ANWEISUNG



Semantik:

Das Rendezvous über den aufgerufenen Eingang wird unmittelbar ausgeführt, falls dies möglich ist, d.h. der gerufene Prozeß an einer entsprechenden accept-Anweisung wartet. Andernfalls werden die auf else folgenden Anweisungen ausgeführt.

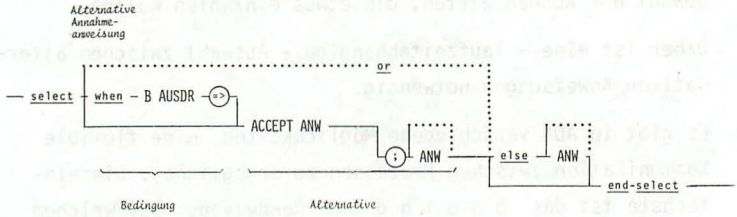
Beispiel:

```
select boss.lager_infos("meldung");
else raeume_auf end select;
```

Auf der anderen Seite kann ein Prozeß gleichzeitig für mehr als einen Eingang ein Rendezvous anbieten:

Syntax: Aus

ADA 24: STRUKTURIERTE ANWEISUNG



Semantik:

Es werden zunächst alle in der Anweisung den Alternativen vorausgehenden Bedingungen ausgewertet. Eine Alternative heißt o f f e n , falls ihr keine Bedingung vorangeht oder diese Bedingung wahr ist, andernfalls heißt die Alternative g e s p e r r t . Sind alle Alternativen gesperrt, so werden die Anweisungen des else-Zweiges ausgeführt, falls dieser vorhanden ist. Fehlt der else-Zweig, so wird die Ausnahme select_error ausgelöst.

Andernfalls wird aus allen offenen Alternativen eine Alternative ausgewählt, für welche ein Eingangsaufruf vorliegt. Kann keine der offenen Alternativen ausgewählt werden, so werden die Anweisungen des else-Zweiges ausgeführt. Fehlt der else-Zweig, so wird auf einen passenden Eingangsaufruf gewartet.

Beispiel:

```
select
   when clock().second in besuchszeit =>
      accept empfang(persoenlich)
                        (gespraech : in geschaeftspartner) do
         notizzettel:= gespraech;
         formular_ausfuellen(notizzettel,formular);
         arbeitsverwaltung.auftragsannahme(formular);
      end empfang;
   or accept empfang(telefonisch)
                        (gespraech : in geschaeftspartner) do
         notizzettel:= gespraech;
      end empfang;
      formular_ausfuellen(notizzettel,formular);
      arbeitsverwaltung.auftragsannahme(formular);
   else null;
end select;
```

Hinweise:

Können mehrere Alternativen ausgewählt werden, so ist nicht festgelegt, welche ausgewählt wird. Diese Auswahl ist rein zufällig und i.a. nicht reproduzierbar. Es ist jedoch sichergestellt, daß nur eine Alternative ausgewählt wird. Man kann also durch eine solche select-Anweisung sicherstellen, daß zwei Eingänge nie gleichzeitig ausgeführt werden.

9.4 DAS VERZÖGERTE RENDEZVOUS

Beim bedingten Rendezvous wurde ein Eingangsaufruf entweder sofort durchgeführt oder überhaupt nicht, wobei im gerufenen Prozeß mehrere Eingangsmöglichkeiten zur Verfügung gestellt werden können. Das v e r z ö g e r t e Rendezvous gibt nun auch die Möglichkeit, eine bestimmte Wartezeit bis zu einer

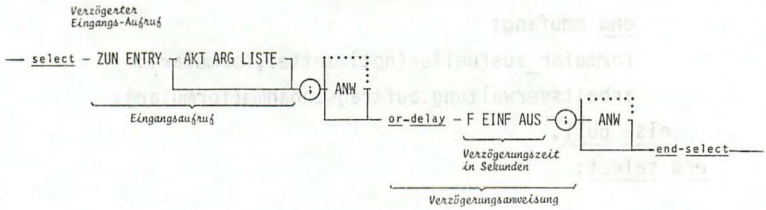
möglichen Synchronisierung vorzuschreiben. Für die Angabe dieser Wartezeit oder sogenannten *Verzögerungszeit* werden Werte des vordefinierten Festpunkttyps duration gemäß der Typdefinition

```
type duration is delta 0.1 range -86400.0 .. 86400.0;
```

verwendet. Werte dieses Typs geben jeweils die Verzögerungszeit in Sekunden (höchstens einen Tag) an. Nichtpositive Verzögerungszeiten haben keine Wirkung.

Syntax: Aus

ADA 24: STRUKTURIERTE ANWEISUNG



Semantik:

Der Ausdruck für die *Verzögerungszeit*, der vom vordefinierten Festkommatyp duration ist, wird ausgewertet. Ist der Eingangsaufruf innerhalb der Verzögerungszeit (in sec.) möglich, so wird er durchgeführt. Andernfalls wird die Anweisungsfolge hinter der Verzögerungsanweisung ausgeführt.

Beispiel:

```
select netzteil.aufladen;
or     delay 300.0;  -- 5 min. Chance
end select;
```

Eine *Verzögerungsanweisung* kann in einem Prozeß auch isoliert auftreten:

Syntax: Aus

ADA 23: ANWEISUNG (ANW)

Verzögerungsanweisung

Semantik:

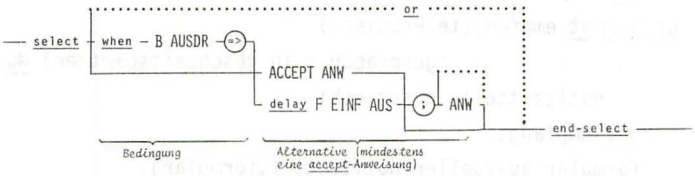
Der Ausdruck für die Verzögerungszeit wird ausgewertet und der Prozeß für die Verzögerungszeit (in sec.) in den Wartezustand versetzt.

Im aufgerufenen Prozeß kann innerhalb einer alternativen accept-Anweisung auch eine Verzögerungsanweisung als Alternative auftreten. Dann darf allerdings kein else-Zweig vorgesehen sein, und mindestens eine Alternative muß mit einer accept-Anweisung beginnen.

Syntax: Aus

ADA 24: STRUKTURIERTE ANWEISUNG

alternative accept-Anweisung



Semantik:

Zur Bestimmung der offenen Alternativen werden alle Bedingungen ausgewertet. Dabei wird für eine offene Alternative mit einer Verzögerungsanweisung unmittelbar nach Auswertung der Bedingung auch der Ausdruck für die Verzögerungszeit ausgewertet. Falls das entsprechende Rendezvous möglich ist,

wird eine offene Alternative mit einer accept-Anweisung ausgewählt. Nach Durchführung des Rendezvous' und der nachfolgenden Anweisungen ist die Ausführung der alternativen accept-Anweisung beendet.

Kann innerhalb der kürzesten Verzögerungszeit aller offenen Alternativen kein Rendezvous durchgeführt werden, so werden die Anweisungen hinter der zugehörigen Verzögerungsanweisung durchgeführt. (Haben zwei Alternativen die gleiche Verzögerungszeit, so ist nicht festgelegt, welche gewählt wird.) Damit ist die Ausführung der alternativen accept-Anweisung beendet.

Beispiel:

```
select
   when clock().second in besuchszeit =>
      accept empfang(persoenlich)
                        (gespraech : in geschaeftspartner) do
         notizzettel:= gespraech;
         formular_ausfuellen(notizzettel,formular);
         arbeitsverwaltung.auftragsannahme(formular);
      end empfang;
   or accept empfang(telefonisch)
                        (gespraech : in geschaeftspartner) do
         notizzettel:= gespraech;
      end empfang;
      formular_ausfuellen(notizzettel,formular);
      arbeitsverwaltung.auftragsannahme(formular);
   or accept lager_infos(nachricht : in string);
   or delay 900.0;
end select;
```

Hinweise:

(1) Es ist nicht sinnvoll, mehrere Alternativen für Verzöge-

rungsanweisungen mit äquivalenten Bedingungen zu versehen, da immer nur die Verzögerungsanweisung mit der kürzesten Verzögerungszeit zum Tragen kommt.

(2) Die Auswahl offener Alternativen für ein Rendezvous ist maschinenabhängig. Programme, die hier auf einen speziellen Auswahlmechanismus aufbauen, sind fehlerhaft. Es sind also folgende Situationen zu vermeiden:
- Eine alternative accept-Anweisung besitzt mehr als eine offene Alternative für einen aufgerufenen Eingang.
- Es wurden verschiedene Eingänge eines Prozesses aufgerufen, und eine accept-Anweisung besitzt offene Alternativen für diese Eingänge.

9.5 DIE TERMINIERUNG VON PROZESSEN

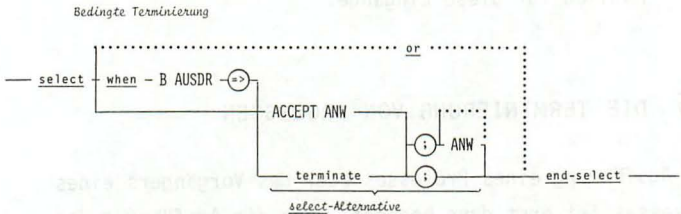
Die Ausführung eines Prozesses oder des Vorgängers eines Prozesses ist erst dann beendet, wenn die Ausführung des zugehörigen Rumpfes beendet ist und alle von ihm abhängigen Prozesse beendet (t e r m i n i e r t) sind. Entsprechend der Aufgabenstellung für Prozesse wird im Regelfall bei der Ausführung des Prozeßrumpfes das Ende des Anweisungsteils nicht erreicht werden. Daher ist für die normale Terminierung auch eine spezielle terminate-Anweisung vorgesehen.

Beispielsweise betrachten wir einen Prozeß π, von dem keine weiteren Prozesse abhängen, d.h. in dessen Vereinbarungsteil keine weiteren Prozesse vereinbart sind. π sei ein passiver Prozeß, der eine Reihe von Dienstleistungen über entsprechende Eingänge anbietet. Die zugehörigen accept-Anweisungen seien in einer unendlichen Schleife realisiert, so daß das Ende des Prozeßrumpfes von π nie erreicht wird. Für seine Terminierung muß dem Prozeß π bekannt sein, daß keiner seiner Eingänge mehr aufgerufen wird. Spätestens, wenn alle Prozesse,

die π aufrufen, und dessen Vorgänger auf ihre Terminierung warten, muß auch π terminiert werden. Dies geschieht mit einer terminate-Alternative innerhalb einer alternativen accept-Anweisung. Denn während der Abarbeitung einer einfachen accept-Anweisung ist es sicher nicht sinnvoll, π zu beenden, während zwischen zwei accept-Anweisungen gar nicht festgestellt werden kann, ob noch ein Eingangsaufruf an π erfolgen wird.

Syntax: Aus

ADA 24: STRUKTURIERTE ANWEISUNG



Semantik:

Die Wahl einer terminate-Alternative bewirkt die normale Terminierung eines Prozesses. Sie kann nicht in einem inneren Block stehen, welcher Prozesse vereinbart.

Eine offene terminate-Alternative kann nur gewählt werden, falls der Prozeß, in dem sie auftritt, zur Menge der von einem Block, Unterprogramm oder Prozeß abhängigen Prozesse gehört, und

- entweder das Ende dieses Blockes, des Unterprogrammrumpfes oder Prozeßrumpfes erreicht ist
- oder im Falle der Abhängigkeit von einem Prozeß eine terminate-Alternative im Prozeßrumpf erreicht ist.

Weiter kann diese terminate-Alternative nur dann gewählt werden, wenn alle anderen Prozesse dieser Menge, wie auch

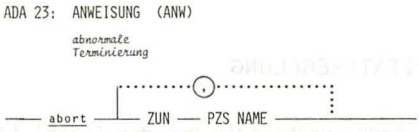
jeder davon abhängige Prozeß (und so weiter) entweder terminiert ist oder an einer alternativen accept-Anweisung mit einer terminate-Alternative wartet.

Beispiele:

```
1. select
      accept lebenszeichen;
      or terminate;
   end select;
2. select
      accept aufladen(ladezeit : duration := 216Ø.Ø) do
         delay ladezeit;
      end aufladen;
      or terminate;
   end select;
```

Neben der normalen Terminierung eines Prozesses durch Auswahl einer terminate-Alternative oder durch Erreichen des Rumpfendes kann ein Prozeß auch von einem anderen Prozeß aus (sozusagen gewaltsam) abgebrochen werden. Man spricht dann von einer a b n o r m a l e n Terminierung.

Syntax: Aus



Semantik:

Die aufgelisteten Prozesse werden terminiert, ebenso alle abhängigen Prozesse. Weiter werden alle Prozesse abnormal terminiert, die von Unterprogrammen abhängen, welche von den terminierten Prozessen direkt oder indirekt aufgerufen wer-

den. Ein abnormal terminierter Prozeß wird aus der Warteschlange für die von ihm gerufenen Eingänge ausgereiht; befindet er sich gerade in einem Rendezvous, wird der rufende Prozeß terminiert, während der gerufene Prozeß das Rendezvous normal durchführt. Falls noch (möglicherweise verzögerte) Eingangsaufrufe für einen abnormal terminierten Prozeß anstehen, so wird die Ausnahme tasking_error für jeden rufenden Prozeß ausgelöst (einschließlich für den Prozeß, der sich gerade im Rendezvous mit diesem befindet) ohne Berücksichtigung einer eventuell vorgesehenen Verzögerungszeit.

Beispiel:

```
exception
   when akku_leer => abort boss;
```

Hinweise:

(1) Die abnormale Terminierung eines Prozesses sollte nur in extremen Fehlersituationen vorgenommen werden. In den meisten Fällen wird das Auslösen der Ausnahme failure für den betroffenen Prozeß ausreichen, wodurch dort fehlerbehandelnde Maßnahmen ermöglicht werden.

(2) Ein Prozeß kann jeden beliebigen Prozeß, auch sich selbst, abnormal terminieren.

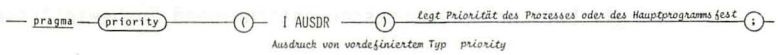
9.6 PRIORITÄTSREGELUNG

Ein Programm kann, unabhängig von der Anzahl tatsächlich zur Verfügung stehender Prozessoren, eine beliebige Anzahl von parallelen Prozessen enthalten. Bei der Ausführung des Programmes kann dies, falls die Anzahl der Prozessoren nicht ausreicht oder überhaupt nur ein Prozessor zur Verfügung steht, die sequentielle Ausführung paralleler Prozesse zur

Folge haben. Da solche Prozesse verschiedene Dringlichkeit haben können. ist es für den Programmierer wichtig, diese selbst festzulegen.

Syntax: Aus

ADA 52: PRAGMA



Semantik:

Die Direktive priority kann in einer Prozeßspezifikation oder im Vereinbarungsteil eines Hauptprogrammes stehen. Sie bewirkt zur Übersetzungszeit die Zuordnung der durch den I AUSDRUCK bestimmten Priorität an den zugehörigen Prozeßtyp bzw. an das Hauptprogramm.

Der vordefinierte Typ priority ist ein implementierungsabhängiger Untertyp von integer. Größere Werte dieses Typs bedeuten höhere Priorität.

Prioritäten kommen dann zur Wirkung, wenn mehr parallele Prozesse zur Ausführung anstehen, als tatsächlich Prozessoren momentan zur Verfügung stehen. In diesem Fall kann bei zwei Prozessen verschiedener Priorität nicht der Fall eintreten, daß der Prozeß mit niederer Priorität ausgeführt wird, während der Prozeß mit höherer Priorität nicht ausgeführt wird.

Bei Prozessen mit gleichen Prioritäten ist eine Ausführungsreihenfolge nicht festgelegt, während bei Prozessen ohne explizite Prioritätszuordnung überhaupt keine Regeln außer für die Zeit eines Rendezvous' festgelegt sind:

Befinden sich zwei Prozesse in einem Rendezvous, so wird das Rendezvous mit der höheren der den Prozessen zugeordne-

ten Prioritäten ausgeführt. Haben beide Prozesse keine zugeordnete Priorität, dann ist auch für das Rendezvous keine Priorität definiert.

Hinweise:

Prioritäten sollten nur verwendet werden, um relative Dringlichkeiten von Prozessen entscheidend zu beeinflussen.

9.7 GEMEINSAME VARIABLE

Die Kommunikation zwischen Prozessen läuft in der Regel über ein entsprechendes Rendezvous ab. Daneben besteht natürlich auch die Möglichkeit der Verwendung derselben globalen Variablen in verschiedenen Prozessen. Dabei entsteht aber die Schwierigkeit, daß ein Prozeß nur zu Zeiten der Synchronisierung mit einem anderen Prozeß genaue Kenntnis darüber hat, ob dieser momentan auf eine gemeinsame Datenmenge zugreift oder nicht. Dabei sind Beginn und Ende eines gemeinsamen Rendzvous' Synchronisierungszeitpunkte für zwei Prozesse und der Zeitpunkt der Aktivierung eines Prozesses Synchronisierungszeitpunkt mit dem aktivierenden Prozeß. Werden also gemeinsame Variable benutzt, so trägt der Programmierer die Verantwortung dafür, daß eine gemeinsame Variable nicht gleichzeitig von mehr als einem Prozeß benutzt oder verändert wird.

Darüberhinaus ist die Realisierung gemeinsamer Variablen durch die Sprache nicht festgelegt. Es ist also durchaus möglich, daß im konkreten Fall für jeden Prozeß Kopien von globalen Variablen angelegt werden und die globale Variable nicht immer auf dem entsprechenden Stand wie diese Kopie gehalten wird. Will man erreichen, daß eine gemeinsame Variable tatsächlich auf ihren neuesten Wert gebracht wird,

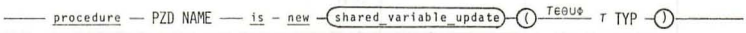
ist für diese Variable eine spezielle Prozedur aufzurufen, welche man als Ausprägung des Standardprozedurschemas shared_variable_update erhält.

```
generic type shared is limited private;
procedure shared_variable_update (x:in out shared);
```

Aus diesem Schema wird mit dem Typ der Variable eine Prozedur erzeugt, die jedesmal dann aufzurufen ist, wenn eine Veränderung der gemeinsamen Variablen vorgenommen wurde. Nur durch den Aufruf dieser Prozedur ist sichergestellt, daß die Änderung auch wirksam wird.

Syntax: Aus

ADA 8: UNTERPROGRAMM ERZEUGUNG



Semantik:

Als Ausprägung erhält man eine Prozedur, durch deren Aufruf man eine für mehrere Prozesse gemeinsame Variable auf ihren neuesten Wert bringt.

Hinweise:

Das Prozedurschema shared_variable_update ist nicht im Paket standard vereinbart, sondern bildet eine eigene Bibliothekseinheit. Deshalb muß es durch eine Kontextangabe bekannt gemacht werden.

9.8 ATTRIBUTE

Folgende Attribute sind für einen Prozeßtyp bzw. Prozeß definiert:

Name des Attributs	Typ des Attributs	Wert
terminated	boolean	falsch, nach Vereinbarung bzw. Allocation des Prozesses, wahr, nach seiner Terminierung
priority	priority	Priorität des Prozesses bzw. Prozeßtyps, falls diese definiert wurde (andernfalls darf das Attribut nicht benutzt werden)
storage_size	universal_integer	Speicherplatzbedarf für den Prozeß bzw. den durch den Prozeßtyp festgelegten Prozeß in Speichereinheiten

Folgendes Attribut für die Eingänge eines Prozesses kann innerhalb des zugehörigen Prozeßrumpfes verwendet werden:

Name des Attributs	Typ des Attributs	Wert
count	integer	Anzahl der auf diesen Eingang wartenden Prozesse

9.9 BEISPIEL *)

Die Programmeinheit grosshandel simuliert einen vereinfachten Handelsbetrieb, der aus den folgenden 3 Abteilungen besteht:

- Chefabteilung (nimmt Aufträge entgegen, bzw. gibt Bestellungen auf - repräsentiert also die Firma gegenüber Geschäftspartnern)
- Organisation (speichert die Aufträge/Bestellungen ab und verteilt die anfallende Arbeit je nach Dringlichkeit oder anderen firmeninternen Prioritäten)
  (nicht zu verwechseln mit dem Paket organisation)
- Versand (führt die Lagerverwaltung durch, d.h. speichert ankommende Waren ab und bedient die Kunden mit den gewünschten Waren)

Diese 3 Abteilungen arbeiten voneinander unabhängig (parallel) und werden demgemäß durch Prozesse realisiert:

Prozeß boss verkörpert die Chefabteilung,
Prozeß arbeitsverwaltung die Organisation (benutzt dabei das Schemapaket schlangenpaket)
und die Prozesse vom Prozeßtyp roboter verkörpern die Lagerarbeiter, also den Versand.

Der Typ roboter taucht dabei nur als Recordkomponente auf. Der Lagerarbeiter wird also eigentlich durch den Typ robbie verkörpert, in dem noch allgemein zugänglich seine persönlichen Daten abgespeichert werden. Bei seiner 'taufe' bekommt er selbst einen lokalen Zeiger auf sich selbst zugewiesen, so daß auch ihm diese persönlichen Daten bekannt werden.

Programmablauf:

Der grosshandel startet zu Beginn durch Aktivierung der Prozesse boss, arbeitsverwaltung und bert.job, ernie.job, fred.-

*) Dieses Beispiel wurde als Seminararbeit von den Herren cand. math. M. Hanke und T. Rössel verfaßt.

job. Während der boss sich gleich in die Arbeit stürzt und die arbeitsverwaltung auf Aufträge wartet (selective wait), werden die 3 Lagerarbeiter erst getauft, bevor sie richtig arbeiten (Aufruf des Eingangs taufe). Der boss empfängt erst Briefe und dann Besucher oder Telefonanrufe und gibt Bestellungen auf oder nimmt sie entgegen (nachts übernimmt diese Arbeit ein Anrufbeantworter). Die Bestellungen bzw. ihre Auswirkungen auf den Versand werden der arbeitsverwaltung weitergegeben, die diese Informationen abspeichert (in der kartei).

Die Lagerarbeiter rufen ihrerseits die arbeitsverwaltung auf, wenn sie den letzten Befehl ausgeführt haben, um neue Befehle zu erhalten.

zusätzliche Einrichtungen:

Prozeß netzteil: Da die roboter bei ihrer Arbeit Strom verbrauchen, müssen sie sich gelegentlich wieder aufladen (am netzteil).
Sollten sie vor lauter Arbeit nicht zum Aufladen kommen, sondern 'ausbrennen', können sie kurz vor dem 'Tod' noch die Firma in die Luft sprengen (exception akku_leer).

Prozeßtyp druckknopf: Um auch eine evtl. Terminierung zu ermöglichen (finanz. Bankrott, etc.) muß eine Vorrichtung existieren, daß die anderen Prozesse wissen, wann der boss terminiert. Dafür wurde der Prozeßtyp druckknopf installiert, der regelmäßig benutzt wird, um einem Kollegen mitzuteilen: "Ich arbeite noch". Sollte also der boss terminieren, erfolgt kein solches Signal mehr bei dem Vorarbeiter bert, also wartet bert auf ein Lebenszeichen bis zum Programmende und analog die anderen Lagerarbeiter, bis schließlich der ganze grossbetrieb terminiert.

```
package calendar is
   type time is
      record year : integer range 1901 .. 2099;
             month : integer range 1 .. 12;
             day : integer range 1 .. 31;
             second : duration;
      end record;
--
   function clock return time;
--
   function "+"(a:time;b:duration) return time;
   function "+"(a:duration;b:time) return time;
   function "-"(a:time;b:duration) return time;
   function "-"(a:time;b:time) return duration;
end calendar;

generic
   type elem is private;
   package schlangenpaket is
      type schlangeninfo is private;
      procedure push(schlange: in out schlangeninfo;e: in
                 elem);
      procedure pop(schlange: in out schlangeninfo;e: out
                 elem);
      function laenge(schlange: in schlangeninfo) return
                 integer;
   private
      type listenelem;
      type listenzeiger is access listenelem;
      type listenelem is
         record datenkomponente : elem;
                next : listenzeiger;
         end record;
      type schlangeninfo is
```

```
        record first,last : listenzeiger:= null;
                    laenge : integer:= Ø;
        end record;
end schlangenpaket;

package body schlangenpaket is
   procedure push(schlange: in out schlangeninfo;e: in
                    elem) is
   begin
   if schlange.last=null
      then schlange.last:= new listenelem(e,null);
           schlange.first:= schlange.last;
      else schlange.last.next:= new listenelem(e,null);
           schlange.last:= schlange.last.next;
   end if;
   schlange.laenge:= schlange.laenge + 1;
   end push;
   procedure pop(schlange: in out schlangeninfo;e: out
                    elem) is
      begin
      e:= schlange.first.datenkomponente;
      schlange.first:= schlange.first.next;
      schlange.laenge:= schlange.laenge - 1;
         -- Das tatsaechliche Programm muesste dafuer sor-
         -- gen, dass die ueberfluessigen Listenelemente
         -- in einer Pointerhalde abgelegt werden, um
         -- spaeter neu verwendet werden zu koennen.
         -- Sonst wuerde das Programm bei einer gut flo-
         -- rierenden Firma mit Sicherheit im Chaos enden.
   end pop;
   function laenge(schlange: in schlangeninfo) return
                    integer is
      begin
      return schlange.laenge;
      end laenge;
end schlangenpaket;
```

```
with calendar, schlangenpaket;
package organisation is
   use calendar;
   subtype besuchszeit is duration range 36000.0 .. 57600.0;
           -- 10 - 16 Uhr
   frueher_morgen : constant duration := 28800.0; -- 8 Uhr
   feierabend : constant duration := 64800.0;  -- 18 Uhr
   type richtung is (eingang,ausgang);
   type geldbetrag is delta 0.01 range 0.00 .. 999_999.99;
   type geschaeft is
      record name : string(1 .. 20);
             warenbezeichnung : natural;
             warenrichtung : richtung;
             preis_pro_einheit : geldbetrag;
             menge : natural;
      end record;
   type auftragstyp is (lieferungen,bestellungen);
   type datenblock is record
                        -- Firmeninterne Warenbeschreibung
                        typ : auftragstyp;
                     end record;
   package kartei is new schlangenpaket (element =>
                                          datenblock);
   use kartei;
   akku_kapazitaet : constant integer := 100;
   subtype ladung is integer range 0 .. akku_kapazitaet;
   existenzminimum : constant ladung := 15;
   energieschub : constant ladung := 5;
end organisation;

with organisation;
procedure grosshandel is
   use organisation;
   type robbie;
```

```
type robbie_zeiger is access robbie;
task type roboter is
   entry taufe(wer_bin_ich : robbie_zeiger);
   entry lebenszeichen;
end roboter;
type robbie is record tag_der_anschaffung : time :=
                        clock();
                      kollege : robbie_zeiger := null;
                      job : roboter;
                  end record;
bert,ernie,fred : constant robbie_zeiger := new robbie;
task boss is
   entry empfang(kontakt)(gespraech: in geschaeft);
   entry lager_infos(nachricht: in string);
end boss;
task arbeitsverwaltung is
   entry auftragsannahme(auftrag:in datenblock);
   entry arbeitsvergabe(auftrag: out datenblock);
end arbeitsverwaltung;
task netzteil is
   entry aufladen(ladezeit: duration := 216Ø.Ø);
end netzteil;
task type druckknopf is
   entry infos(wen_soll_ich_rufen: robbie_zeiger);
end druckknopf;
type ruf is access druckknopf;
task body arbeitsverwaltung is
   news,aktueller_auftrag : datenblock;
   aktueller_auftrag_vorhanden : boolean := false;
   warteschlange : array(auftragstyp) of schlangeninfos;
   begin
      loop
         select
            accept auftragsannahme(auftrag: in
                                          datenblock) do
```

```
                news:= auftrag;
                end auftragsannahme;
            if not aktueller_auftrag_vorhanden
            then aktueller_auftrag:= news;
                 aktueller_auftrag_vorhanden:= true;
            else push(warteschlange(news.typ),news);
            end if;
        or when aktueller_auftrag_vorhanden =>
            accept arbeitsvergabe(auftrag: out
                                        datenblock) do
                auftrag:= aktueller_auftrag;
                end arbeitsvergabe;
            if laenge(warteschlange(lieferungen)) = Ø
               and laenge(warteschlange(bestellungen))
                                                  = Ø
            then aktueller_auftrag_vorhanden:= false;
            else null;
                -- hier wird aktueller_auftrag mit Hilfe
                -- der Prozedur pop (aus kartei) - abhaen-
                -- gig von firmeninternen Prioritaeten -
                -- mit dem jeweils ersten Auftrag aus der
                -- Bestellungs- bzw. Lieferungsschlange
                -- belegt.
            end if;
        or terminate;
        end select;
    end loop;
end arbeitsverwaltung;
task body netzteil is
  begin
    loop
        select accept aufladen(ladezeit: duration :=
                          216Ø.Ø) do
                          delay ladezeit;
```

```
                end aufladen;
            or terminate;
            end select;
        end loop;
    end netzteil;
task body druckknopf is
    arbeiter : robbie_zeiger := null;
    begin
        accept infos(wen_soll_ich_rufen: robbie_zeiger) do
            arbeiter:= wen_soll_ich_rufen;
        end infos;
        if arbeiter /= null then arbeiter.job.lebenszeichen;
        end if;
    end druckknopf;
task body roboter is
    aktuelle_ladung : ladung := akku_kapazitaet;
    akku_leer : exception;
    eigene_person : robbie_zeiger := null;
    signal : ruf := null;
    auftrag : datenblock;
    begin
        accept taufe(wer_bin_ich: robbie_zeiger) do
            eigene_person:= wer_bin_ich;
        end taufe;
        signal:= new druckknopf; signal.infos(null);
        lebenslauf:
        loop
            select arbeitsverwaltung.arbeitsvergabe
                                        (auftrag);
                -- Robbie versucht, den Auftrag auszufüh-
                -- ren. Wenn etwas nicht machbar ist (Wa-
                -- renmangel, Warenueberschuss etc.), er-
                -- folgt ein verzoegerter Aufruf von
                -- boss.lager_infos("Meldung"). Sinkt
```

```
                -- Robbies Akku-Ladung waehrend der Arbeit
                -- auf Null ab, wird die Ausnahme akku_leer
                -- ausgeloest.
                if aktuelle_ladung < existenzminimum
                then select netzteil.aufladen;
                     or     delay 300.0;  -- 5 min. Chance
                     end select;
                end if;
            else if aktuelle_ladung < akku_kapazitaet
                                          -- Energieschub
                then select netzteil.aufladen;
                     or delay 60.0;  -- 1 min. Chance
                     end select;
                end if;
            end select;
            select accept lebenszeichen;
            or terminate;
            end select;
            if signal.all'terminated
            then signal:= new druckknopf;
                 signal.infos(eigene_person.kollege);
            end if;
        end loop lebenslauf;
        exception
            when akku_leer => abort boss;
                              -- Robbie bez. Energiereserven
                              -- ausgeblutet. Unerbittliche
                              -- Bestrafung des Chefs.
    end roboter;
task body boss is
    notizzettel,cassette : geschaeft;
    signal : ruf := null;
    vorarbeiter : robbie_zeiger := bert;
    formular : datenblock;
```

```
bankrott,reich_genug : boolean := false;
procedure formular_ausfuellen
          (notiz: in geschaeft; formular: out
          datenblock) is
   begin
      null; -- Gespraech zu firmeninternen Daten auf-
            -- bereiten.
   end formular_ausfuellen;
begin
   signal:= new druckknopf;
   signal.infos(null);
   tagesablauf:
   loop
      post_durchsehen: while empfang(schriftlich)'count
                                                  > Ø
      loop
         accept empfang(schriftlich)
                (gespraech: in geschaeft) do
            notizzettel:= gespraech;
            -- Je nach Lage der Kontoauszuege kann
            -- hier bankrott bzw. reich_genug auf true
            -- gesetzt werden.
         end empfang;
         formular_ausfuellen(notizzettel,formular);
         arbeitsverwaltung.auftragsannahme(formular);
      end loop post_durchsehen;
      tagsueber:
      loop
         if signal.all'terminated
         then signal:= new druckknopf;
              signal.infos(vorarbeiter);
         end if;
         select
            when clock().second in besuchszeit =>
```

```
        accept empfang(persoenlich)
                (gespraech: in geschaeftspartner)
                do
            notizzettel:= gespraech;
            formular_ausfuellen(notizzettel,
                                formular);
            arbeitsverwaltung.auftragsannahme(
                                formular);
        end empfang;
    or accept empfang(telefonisch)
            (gespraech: in geschaeftspartner) do
            notizzettel:= gespraech;
        end empfang;
        formular_ausfuellen(notizzettel,formular);
        arbeitsverwaltung.auftragsannahme
                                    (formular);
    or accept lager_infos(nachricht: in string);
        -- In Abhaengigkeit von der Nachricht Ware
        -- bestellen oder abbestellen, bei Ueber-
        -- lastung des Lagers weitere Roboter an-
        -- schaffen.
    or delay 900.0;  -- boss guckt alle 15 min.
        -- auf die Uhr! Mit Sekretaerin flirten,
        -- den klugen Kopf hinter der dazugehoe-
        -- rigen Zeitung verschwinden lassen, oder
        -- alles, was Maennern Spass macht, lesen.
    end select;
    exit when clock().second >= feierabend;
end loop tagsueber;
nachts:
loop  -- Anrufbeantworter tut das Noetigste
    if signal.all'terminated
    then signal:= new druckknopf;
         signal.infos(vorarbeiter);
    end if;
```

```
            select
                accept empfang(telefonisch)
                        (gespraech: in geschaeft) do
                        cassette:= gespraech;
                end empfang;
                formular_ausfuellen(cassette,formular);
                arbeitsverwaltung.auftragsannahme(formular);
            else null; -- busy looping around
            end select;
            exit when clock().second  -- Wenn Chef kommt,
                                      -- +- 5 min.
                in frueher_morgen-300.0 .. frueher_morgen+
                    300.0;
        end loop nachts;
        exit when bankrott or reich_genug;
    end loop tagesablauf;
    -- Jetzt hat's ein Ende mit der Schufterei!
  end boss;
begin  -- Ab hier arbeitet der grosshandel, also
    -- boss, arbeitsverwaltung und die roboter.
  bert.kollege:= ernie;
  bert.job.taufe(bert);
  ernie.kollege:= fred;
  ernie.job.taufe(ernie);
  fred.job.taufe(fred);
end grosshandel;
```

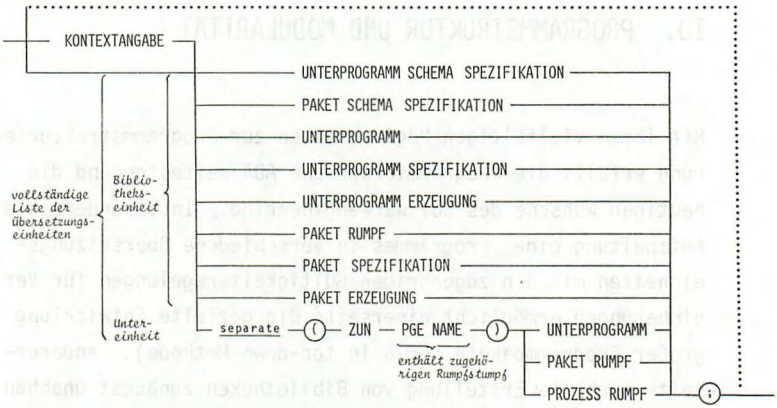
10. PROGRAMMSTRUKTUR UND MODULARITÄT

Mit ihren vielfältigen Möglichkeiten zur Programmstrukturierung erfüllt die Programmiersprache ADA weitestgehend die heutigen Wünsche des Softwareengineering. Insbesondere die Aufspaltung eines Programmes in verschiedene Übersetzungseinheiten mit den zugehörigen Gültigkeitsregelungen für Vereinbarungen ermöglicht einerseits die gezielte Entwicklung großer Programmpakete (etwa in top-down-Methode), andererseits auch die Erstellung von Bibliotheken zunächst unabhängiger, allgemein brauchbarer Routinen, welche zu einem späteren Zeitpunkt erst zu größeren Programmen zusammengefaßt werden (bottom-up-Methode).

10.1 DIE PROGRAMMBIBLIOTHEK

Ein Programm besteht aus einer oder mehreren *Übersetzungseinheiten*, die gemeinsam oder getrennt übersetzt werden. Die Übersetzungseinheiten eines Programmes gehören zu einer P r o g r a m m b i b l i o t h e k . Jede Übersetzungseinheit, die nicht *Untereinheit* einer anderen ist, heißt B i b l i o t h e k s e i n h e i t .

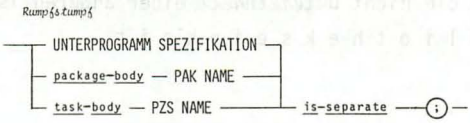
ADA 1: PROGRAMM



Zur Identifikation müssen innerhalb einer Programmbibliothek die Namen aller Bibliothekseinheiten verschieden sein; dies gilt natürlich nicht für den zu einer entsprechenden Spezifikation gehörigen Rumpf, die beide denselben Namen führen müssen. Der Name eines getrennt übersetzten Unterprogrammes darf kein Operatorsymbol sein; eine getrennt übersetzte Funktion kann aber in einen Operator umbenannt werden.

Ein Unterprogramm, ein Paketrumpf oder ein Prozeßrumpf, innerhalb einer übergeordneten Einheit vereinbart, kann als eine *Untereinheit* definiert und somit getrennt von ihrer übergeordneten Einheit übersetzt werden. In diesem Fall wird in der übergeordneten Einheit an Stelle des Unterprogramm-, Paket- oder Prozeßrumpfes lediglich ein entsprechender *Rumpfstumpf* vereinbart:

Aus ADA 17:



Die zugehörige Untereinheit, welche den vollständigen Rumpf enthält (einschließlich der Spezifikation bei einem Unterprogramm), erwähnt nach dem Wortsymbol separate den Namen der Programmeinheit, in welcher der zugehörige Rumpfstumpf steht.

Ist diese Programmeinheit selbst eine Untereinheit, so muß ihr Name zusammen mit dem Namen ihrer Obereinheit angegeben werden, usw. In diesem Fall ist also die Verwendung eines sogenannten Zunamens für die Programmeinheit notwendig, der die Namen aller übergeordneten Einheiten bis zur ersten Bibliothekseinheit umfaßt (vgl. ADA 58 und 10.3,10.4,10.5). Die Namen aller Untereinheiten einer Bibliothekseinheit und die Namen aller Untereinheiten dieser Einheiten, usw. müssen verschieden sein.

Bezüglich der Gültigkeit und Sichtbarkeit von Größen (siehe 10.2,10.3) wird eine Untereinheit so gehandhabt, als stünde sie tatsächlich an der Stelle des zugehörigen Rumpfstumpfes.

Beispiel:

In dem Paket studienordnung

```
with student;
package studienordnung is
   use student;
   function klausur (leistung:vorbereitung) return note;
   function diplom (leistung:vorbereitung) return boolean;
   function bafoeg (leistung:note) return geld;
end studienordnung;

package body studienordnung is
   function klausur (leistung:vorbereitung) return note is
                                                separate;
   function diplom (leistung:vorbereitung) return boolean is
                                                separate;
```

```
   function bafoeg (leistung:note) return geld is separate;
end studienordnung;
```

sind die Funktionen klausur, diplom, bafoeg nur mit Rumpfstümpfen vereinbart. In der Programmbibliothek sind daher die entsprechenden Untereinheiten erforderlich:

```
separate (studienordnung)
function klausur (leistung:vorbereitung) return note is
begin
   case leistung is
      when uebertrieben => return 1;
      when gut => return 2;
      when ausreichend => return 3;
      when schlecht => return 4;
      when falsch => return 5;
   end case;
end klausur;

separate (studienordnung)
function diplom (leistung:vorbereitung) return boolean is
begin
   return leistung /= falsch;
end diplom;

separate (studienordnung)
function bafoeg (leistung:note) return geld is
begin
   if leistung < 5
      then return (540,0);  -- Der Typ geld wurde in
      else return (0,0);    -- 2.1.2.2 vereinbart
   end if;
end bafoeg;
```

10.2 GÜLTIGKEITSBEREICHE VON VEREINBARUNGEN

Eine Vereinbarung verbindet einen Namen mit einer Größe des Programmes (z.B. einer Variable, einer Record-Komponente, einem Typ, einem Unterprogramm, einem formalen Parameter, usw.). Der Textbereich, in dem diese Vereinbarung eine Wirkung hat, heißt G ü l t i g k e i t s b e r e i c h der Vereinbarung; er beginnt mit der Einführung eines Namens (innerhalb einer Übersetzungseinheit). Da ein Name in verschiedenen Vereinbarungen auftauchen kann, sind Überschneidungen einzelner Gültigkeitsbereiche möglich (siehe hierzu 10.5).

Die Gültigkeitsbereiche von Vereinbarungen liegen folgendermaßen fest:

Größe des Programmes	Ort der impliziten oder expliziten Vereinbarung	Gültigkeitsbereich
beliebig	Vereinbarungsteil eines - Blocks - Unterprogrammrumpfes - Paketrumpfes - Prozeßrumpfes sichtbarer oder geschützter Teil einer Paketspezifikation	bis zum Ende des - Blocks - Unterprogrammrumpfes - Paketrumpfes - Prozeßrumpfes - der Paketspezifikation und ihrem zugehörigen Paketrumpf
Eingang	Prozeßspezifikation	Prozeßspezifikation und zugehöriger Prozeßrumpf
getrennt übersetztes Unterprogramm oder Paket	——	Übersetzungseinheit selbst mit zugehörigem Rumpf und alle ihre Untereinheiten

Recordkomponente bzw. Diskriminante	Komponenten- bzw. Diskriminantenvereinbarung in Recordtypvereinbarung	bis zum Ende des Gültigkeitsbereichs der Recordtypvereinbarung
formales Argument	formale Argumentliste eines - Unterprogramms - Eingangs - Schemas	bis zum Ende des Gültigkeitsbereichs des - Unterprogramms - Eingangs - Schemas
Schleifenparameter (Laufvariable)	Schleife	bis zum Ende der Schleife
Literal eines Aufzählungstyps	Typvereinbarung des Aufzählungstyps	bis zum Ende des Gültigkeitsbereichs der Typvereinbarung

10.3 SICHTBARKEIT VON GRÖSSEN EINES PROGRAMMS

Kann eine Programmgröße innerhalb des Gültigkeitsbereichs der zugehörigen Vereinbarung direkt mit seinem Namen angesprochen werden, so heißt die Größe, seine Vereinbarung und sein Name an dieser Stelle d i r e k t s i c h t b a r . Die folgende Tabelle legt die Sichtbarkeitsbereiche für die Größen eines Programmes in Abhängigkeit vom Ort der Vereinbarung fest. Allerdings ist hier zu beachten, daß die Verwendung desselben Namens in verschiedenen Vereinbarungen möglicherweise zu Überlagerungen führt, aufgrund deren die Sichtbarkeitsregelungen noch etwas modifiziert werden müssen (siehe 10.5).

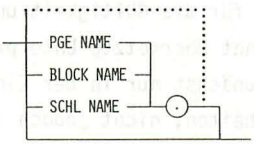
Größe des Programmes	Ort der Vereinbarung	direkt sichtbar in	Außerhalb des Gültigkeitsbereichs der Vereinbarung sichtbar durch
beliebig	Vereinbarungteil im - Block - Unterprogrammrumpf - Paketrumpf - Prozeßrumpf	- Block - Unterprogrammrumpf - Paketrumpf - Prozeßrumpf	——
beliebig	- sichtbarer Teil - geschützter Teil eines Pakets	- Paketspezifikation und zugehörigem -rumpf - geschützter Teil der Paketspezifikation und im zugehörigen Rumpf	Voranstellen des Paketnamens (Zuname) bzw. direkt sichtbar bei Verwendung der use-Klausel ——
Entry	Prozeßspezifikation	Prozeßspezifikation und zugehöriger Prozeßrumpf	Voranstellen des Prozeßnamens (Zuname)
getrennt übersetztes Unterprogramm oder Paket	——	der Übersetzungseinheit selbst und allen zugehörigen Untereinheiten, sowie in allen Übersetzungseinheiten, bei denen diese in einer with-Anweisung aufgeführt ist.	

Recordkomponente	Recordtypvereinbarung	Recordtypvereinbarung und zugehöriger Darstellungsangabe	Voranstellen des Recordnamens
Diskriminante	Recordtypvereinbarung	Diskriminantenteil und zugehöriger Recordtypvereinbarung	Voranstellen des Recordnamens
formales Argument	formale Argumentliste - eines Unterprogrammes - eines Einganges - eines Schemas	- formale Argumentliste und gesamtes Unterprogramm - formale Argumentliste, zugehörige accept-Anweisung - formale Argumentliste, Spezifikation und Rumpf	sichtbar in der Parameterzuordnung einer aktuellen Argumentliste
Schleifenparameter (Laufvariable)	Schleife	ganze Schleife	——
Literal eines Aufzählungstyps	Vereinbarung des Aufzählungstyps	Gültigkeitsbereich der Typvereinbarung	——

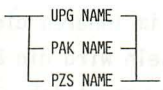
Die obige Tabelle gibt in verschiedenen Fällen durch den sogenannten ZUNAMEn die Möglichkeit, eine Größe außerhalb des Gültigkeitsbereiches ihrer Vereinbarung sichtbar zu machen.

Syntax:

ADA 58: ZUNAME (ZUN)



ADA 59: PROGRAMMEINHEITSNAME (PGE NAME)



Semantik:

Eine Größe kann in der Programmeinheit bzw. in dem Block bzw. in der Schleife, in der sie unmittelbar (also nicht in einem inneren Konstrukt) vereinbart ist, durch Voranstellen des entsprechenden Namens der Programmeinheit bzw. des Blockes bzw. der Schleife angesprochen werden. Entsprechend der Stufe, auf welcher eine Größe innerhalb der Programmeinheit auftritt, kann der Zuname aus allen Namen der umfassenden Konstrukte bestehen.

Hinweise:

Die Verwendung eines Zunamens ist insbesondere dann sinnvoll, wenn in einem inneren Konstrukt der Name der Größe überdeckt und damit die Größe nicht mehr direkt sichtbar ist (siehe 10.5).

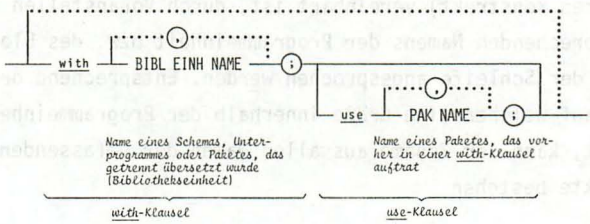
Große Bedeutung besitzt der Zuname für die Bezeichnung von Eingängen im Eingangsaufruf eines Prozesses (vgl. 9.2.2) und für die Bezeichnung von Größen eines mit einer with-Klausel sichtbar gemachten Paketes innerhalb anderer Bibliothekseinheiten (vgl. 10.4).

10.4 KONTEXTANGABEN

Entsprechend den Regeln für die Gültigkeit und Sichtbarkeit von Größen können getrennt übersetzte Unterprogramme und Pakete und deren Größen zunächst nur in der Einheit selbst und in zugehörigen Untereinheiten, nicht jedoch in anderen Bibliothekseinheiten angesprochen werden. Eine modulare Programmierung wäre hierdurch weitgehend unmöglich gemacht. Um diesem Mangel abzuhelfen, kann in einer zugehörigen Kontextangabe durch with-Klauseln die Umgebung einer Übersetzungseinheit festgelegt werden, die im Inneren dieser Einheit benutzt wird, und durch use-Klauseln wird die Bezeichnung der Größen aus dieser Umgebung ganz wesentlich vereinfacht.

Syntax:

ADA 2: KONTEXTANGABE



Semantik:

Durch die with-Klausel wird der Name der (getrennt übersetzten) Bibliothekseinheit sichtbar. Alle in dieser Einheit vereinbarten Größen werden zugänglich, so, als ob eine Vereinbarung der Bibliothekseinheit am Ende des Paketes standard erfolgte.

Falls die in der with-Klausel stehende Bibliothekseinheit

- ein Schema ist, können in der gegebenen Übersetzungseinheit Ausprägungen dieses Schemas vorgenommen werden (siehe 6.).

- ein Unterprogramm ist, so kann dieses aufgerufen werden (siehe 5.).
- ein Paket ist, so können die dort im sichtbaren Teil vereinbarten Größen mit dem Paketnamen als Zuname angesprochen werden (vgl. 1.2, 10.3).

Erscheint der Paketname auch in einer use-Klausel, so brauchen die Größen nicht mit dem Paketnamen als Zuname versehen zu werden, sondern können direkt mit ihrem eigentlichen Namen bezeichnet werden, falls die Eindeutigkeit gewährt ist. Kontextangaben können nicht zur Überdeckung von Namen führen (siehe 10.5).

Hinweise:

(a) Alle vordefinierten Bibliothekseinheiten, außer dem Paket standard, welches stets sichtbar ist, müssen durch with-Klauseln bekannt gemacht werden, falls sie verwendet werden.

(b) Es ist gleichgültig, ob eine Bibliothekseinheit einmal oder öfter in einer Kontextangabe aufgelistet ist.

(c) Die Reihenfolge in with-Klauseln braucht nicht mit der tatsächlichen Verarbeitungsreihenfolge übereinzustimmen.

(d) Bei der Verarbeitung einer Bibliothekseinheit werden die dort in with-Klauseln stehenden Bibliothekseinheiten in einer Reihenfolge herangezogen, welche die Übersetzung des Programms als Ganzes ermöglicht.

(e) use-Klauseln können auch als eigenständige Vereinbarungen im Vereinbarungsteil von Programmeinheiten auftreten (vgl. ADA 18). Entsprechend den Gültigkeitsbereichen der Vereinbarungen können also innerhalb einer Bibliothekseinheit an verschiedenen Stellen die Größen verschiedener Pakete direkt sichtbar sein.

Beispiel: aus 1.1, Version 4:

```
with Standard_Funktionen; use Standard_Funktionen;
   function f is
        :
with Standard_Funktionen; use Standard_Funktionen;
   function fs is
        :
with f,fs; with Standard_EA; use Standard_EA;
procedure newton is
   begin
      get (xk1);
      get (eps);
iteration: loop
            xk:= xk1;
            xk1:= xk - f(xk)/fs(xk);
            exit when abs ((xk1 - xk)/xk) < eps;
         end loop iteration;
end newton;
```

Die Funktionen f, fs werden hier als Bibliothekseinheiten übersetzt und durch eine entsprechende Kontextangabe im Unterprogramm newton bekannt gemacht.

Darüber hinaus werden die Pakete Standard_Funktionen in f bzw. fs und Standard_EA in newton eingeführt; damit sind in f bzw. fs die Größen des sichtbaren Teils von Standard_Funktionen und darüber hinaus in newton auch die Größen des sichtbaren Teils von Standard_EA sichtbar.

10.5 KONFLIKTE BEI DER NAMENSGEBUNG

Derselbe Name kann durchaus in verschiedenen Vereinbarungen eines Programmes eingeführt werden und auch zur Bezeichnung verschiedener Größen verwendet werden. Hierbei treten keinerlei Schwierigkeiten auf, solange sich die Gültigkeitsbereiche entsprechender Vereinbarungen nicht überschneiden (vgl. 10.2).

Überschneiden sich die Gültigkeitsbereiche von zwei Vereinbarungen zu demselben Namen, so unterscheidet man zwischen einem Überladen bzw. einem Überdecken eines Namens.

Das Ü b e r l a d e n eines Namens ist nur bei Unterprogrammnamen (vgl. 5.7) und Aufzählungstypen möglich (vgl. 2.1.1.1.1).

Dabei spricht man von der Überladung eines Unterprogrammnamens, wenn zu einem Namen mehrere Unterprogramme vereinbart sind, deren Spezifikationen nicht äquivalent sind. Bei einem Aufzählungsliteral kommt es lediglich darauf an, daß dieses Literal in verschiedenen Aufzählungstypen auftritt.

Solche überladene Namen können also verschiedene Bedeutungen besitzen, die jedoch durch den Kontext eindeutig auflösbar sein müssen. Überladungen dürfen durch Vereinbarungen ein und desselben Vereinbarungsteils entstehen.

Das Ü b e r d e c k e n eines Namens entsteht durch die Verwendung dieses Namens in zwei Vereinbarungen, die nicht zur Überladung des Namens führen. Diese Vereinbarungen dürfen nicht auf gleicher Stufe im Vereinbarungsteil einer Einheit stehen. Grundsätzlich wird die Bedeutung des Namens gemäß der äußeren Vereinbarung überdeckt, und die zugehörige (g l o b a l e) Größe ist im Gültigkeitsbereich der inneren Vereinbarung , d.h. im Sichtbarkeitsbereich der l o k a l e n Größe nicht mehr direkt sichtbar.

Das Überdecken von Namen wird insbesondere durch die Schachtelung von Unterprogrammen, Paketen, Prozessen und Blöcken auftreten. Dabei wird die formale Schemaargumentliste, die formale Argumentliste eines Unterprogrammes und der Vereinbarungsteil des Unterprogrammrumpfes als ein Vereinbarungsteil aufgefaßt. In ähnlicher Weise gelten formale Schemaargumentliste, Paketspezifikation und Vereinbarungsteil des Paketrumpfes als ein Vereinbarungsteil.

Eine überdeckte Größe, welche unmittelbar in einem Unterprogramm, Paket oder Prozeß vereinbart ist, d.h. lokale Größe einer solchen Einheit ist, wird immer sichtbar durch Angabe des Namens des Unterprogrammes, Paketes oder Prozesses als Zuname der Größe (siehe 10.3).

Letztlich bleibt die Frage zu beantworten, in welcher Weise Namenskonflikte zwischen einer lokalen Größe einer Bibliothekseinheit und den durch Kontextangaben direkt sichtbar gemachten Größen anderer Bibliothekseinheiten gelöst werden. An einer beliebigen Stelle einer Bibliothekseinheit kommen für solche Konflikte alle Größen von solchen Paketen in Betracht, welche in einer an dieser Stelle gültigen <u>use</u>-Klausel aufgeführt sind. Es gilt dann:

- Eine Größe wird direkt sichtbar, falls sie im sichtbaren Teil genau eines solchen Paketes steht und nicht durch eine lokale Vereinbarung überdeckt wird.
- Ein Aufzählungsliteral wird genau dann direkt sichtbar, falls dessen Name nicht schon anderweitig direkt sichtbar ist, und in jedem Fall, wenn es ein Zeichenliteral (z.B. '+', 'a', usw.) ist.
- Ein Unterprogramm wird direkt sichtbar, wenn dessen Name durch keine Größe - lokal vereinbart oder nach den beiden ersten Regeln - überdeckt wird und die Unterprogrammspezifikation von jeder anderen sichtbaren verschieden ist.

Die Verwendung von use-Klauseln führt also in keinem Fall zu Schwierigkeiten, da die Größen sichtbar gemachter Pakete nachrangig direkt sichtbar werden und somit direkt sichtbare lokale Größen nicht überdeckt werden.

Hinweise:

Namenskonflikte lassen sich auf einfache Art und Weise durch die Umbenennung entsprechender Größen (siehe 2.8) oder durch die Verwendung von Zunamen (siehe 10.3) umgehen.

Speziell bei Werten aus Aufzählungstypen können Mehrdeutigkeiten, die durch Überladen entstanden sind, mittels der Qualifizierung dieser Werte in der Form des qualifizierten Ausdruckes (siehe ADA 31, Abschnitt 3.10) ausgeschlossen werden.

10.6 ÜBERSETZUNG UND AUSFÜHRUNG EINES PROGRAMMES

Die Übersetzung eines Programmes kann in mehreren Schritten erfolgen, wobei jeweils in einem Schritt eine Folge von Übersetzungseinheiten übersetzt wird.

Die Reihenfolge der Übersetzung von Übersetzungseinheiten eines Programmes ergibt sich aus den Sichtbarkeitsregeln:

- Eine Einheit darf erst übersetzt werden, nachdem alle Einheiten, deren Namen in einer with-Klausel auftauchen, übersetzt sind.
- Ein Unterprogramm oder ein Paketrumpf darf erst nach der zugehörigen Unterprogrammspezifikation bzw. Paketspezifikation übersetzt werden.
- Untereinheiten dürfen erst nach der zugehörigen Obereinheit übersetzt werden.

Ansonsten ist die Reihenfolge beliebig.

Die Wirkung der Übersetzung einer Übersetzungseinheit ist die Vereinbarung der Einheit als Bibliothekseinheit (bei UNTERPROGRAMM SCHEMA SPEZIFIKATION, PAKET SCHEMA SPEZIFIKATION, UNTERPROGRAMM SPEZIFIKATION, UNTERPROGRAMM ERZEUGUNG, PAKET SPEZIFIKATION, PAKET ERZEUGUNG) bzw. die Vereinbarung der Einheit und Festlegung des zugehörigen Rumpfes.

Werden Änderungen in einer Übersetzungseinheit vorgenommen, so werden diese Einheit und jede Übersetzungseinheit, in welche sich diese Änderungen auswirken könnten, neu übersetzt. Dies können eventuell vorhandene Untereinheiten oder Übersetzungseinheiten sein, in deren with-Klauseln die geänderte Einheit genannt wird. Eine Implementierung darf die Neuübersetzung auf die tatsächlich betroffenen Übersetzungseinheiten beschränken.

Speziell Änderungen in einer Untereinheit oder einem Unterprogramm- bzw. Paketrumpf wirken sich nicht auf andere Übersetzungseinheiten aus, so daß hier nur diese Einheit neu übersetzt wird.

Die A u s f ü h r u n g eines Programmes beginnt mit der Ausführung eines Unterprogrammes, welches die übliche Funktion eines Hauptprogrammes übernimmt. Dabei legt die Sprache nicht fest, wie ein Unterprogramm als Hauptprogramm ausgezeichnet wird. Vor der Ausführung des Hauptprogrammes werden alle durch das Hauptprogramm benutzten Bibliothekseinheiten v e r a r b e i t e t . Dies sind die in den with-Klauseln des Hauptprogrammes genannten Einheiten und deren Untereinheiten, sowie die in deren with-Klauseln genannten Einheiten und Untereinheiten, und so weiter. Die Verarbeitung der Bibliothekseinheiten läuft so ab, daß eine Bibliothekseinheit bereits verarbeitet sein muß, wenn eine andere, davon abhängige Bibliothekseinheit, verarbeitet werden soll. Falls verschiedene Reihenfolgen für die Verarbeitung möglich sind, so darf die Bedeutung des Programmes nicht von einer bestimmten Reihenfolge abhängen.

11. EIN- BZW. AUSGABE UND HANDHABUNG VON FILES

Die Ein- bzw. Ausgabe eines Programms beschreibt den Transfer von Daten von bzw. zu einem externen Datenträger oder Gerät (sie ist daher stets maschinenabhängig). In der Regel interessiert den Benutzer nur die Schnittstelle seines Programms, wie und wo die Daten abgespeichert werden, ist ihm dagegen weitgehend gleichgültig.

Die Ein- und Ausgabe sind wegen der Maschinenabhängigkeit nicht ein Bestandteil des Sprachkerns, d.h. sie sind nicht im Paket standard beschrieben, welches zu jedem ADA-Programm stets automatisch hinzugefügt wird. Sondern es stehen drei vordefinierte Pakete zur Verfügung. Im einzelnen sind dies das Paketschema input_output, das Paket text_io und das Paket low_level_io.

Diese Pakete stellen nicht nur Unterprogramme für Ein/Ausgabe zur Verfügung, sondern beschreiben auch die Files (Dateien), die zur externen Aufbewahrung von Daten verwendet werden.

Files sind eine zusätzliche Klasse von Typen. Sie beschreiben eine Folge von Daten gleichen Typs, also eine beliebig große Anzahl von Komponenten eines Typs.

In den folgenden Diagrammen sind die in Klammern angegebenen Kommentare (*file*), (*item*), (*to*), (*name*), (*spacing*), (*width*), (*mantissa*), (*exponent*), (*fract*), (*lower_case*) die Namen der formalen Argumente.

Da man mit solchen Files auch strukturierte Datentypen bilden kann und sie z.B. als Argument in eine Prozedur eingegeben werden können, wurden sie in die Syntaxdiagramme eingearbeitet.

Man beachte, daß man Files nur vereinbaren kann, falls das entsprechende Ein/Ausgabepaket sichtbar ist.

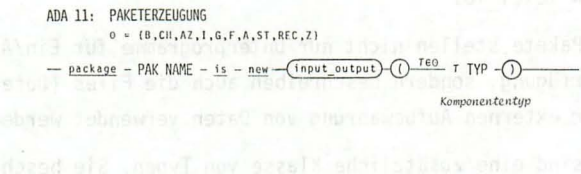
In Abschnitt 4.3 wurden bereits die einfachsten und wichtigsten Ein- und Ausgabeprozeduren besprochen.

11.1 DAS PAKETSCHEMA INPUT_OUTPUT

Das Paketschema input_output beschreibt die Ein- und Ausgabe auf Files von einem festgelegten Typ. Die Werte werden in der internen Form abgelegt (Es kann jedoch ein Unterschied zwischen der internen Form in einem Programm und auf einer Datei bestehen). Der Komponententyp wird bei der Erzeugung von Inkarnationen des Schemas als aktuelles Schemaargument angegeben.

Syntax:

Aus



Semantik:

Es wird ein Ein/Ausgabepaket für den Komponententyp *T* erzeugt. Files mit diesem Komponententyp können nun vereinbart werden. Unterprogramme für Ein- und Ausgabe, sowie zur Verwaltung von Files vom Komponententyp *T* stehen zur Verfügung.

11.1.1 DIE VERWALTUNG VON FILES

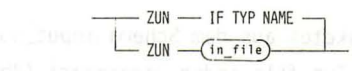
Mit der Erzeugung eines Paketes für Ein- und Ausgabe auf Files mit Komponententyp *T* werden die Filetypen in_file, out_file und inout_file vereinbart.

type in_file is limited private;

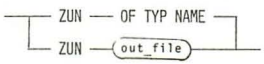
type out_file is limited private;
type inout_file is limited private;

Syntax:

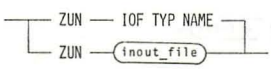
ADA 22: IF TYP (*Eingabedateityp*)



ADA 22: OF TYP (*Ausgabedateityp*)



ADA 22: IOF TYP (*Ein/Ausgabedateityp*)



Semantik:

Es handelt sich hierbei um limitierte geschützte Typen für die Beschreibung von Files mit Komponententyp *T*, die nur für die Eingabe (in_file) (Abkürzung IF), nur für die Ausgabe (out_file) (OF) oder für Ein- und Ausgabe (inout_file) (IOF) vorgesehen sind.

Als Zuname wird in der Regel der Name des erzeugten Ein/Ausgabepaketes angegeben werden.

Variable von diesen Typen, die nun vereinbart werden können, wirken in Ausdrücken wie Konstante, da keine Wertzuweisung möglich ist. Sie können ihren Wert jedoch durch einzelne Anweisungen verändern.

Filetypen dürfen auch als Komponententypen strukturierter Typen auftreten.

Hinweis:

Im Gegensatz zu PASCAL kann in ADA schon zur Übersetzungs-

zeit geprüft werden, ob ein unerlaubter Lesezugriff (auf out_files) oder Schreibzugriff (auf in_files) erfolgt. Außerdem gibt es durch die inout_files die Möglichkeit, ein File "gleichzeitig", d.h. ohne Zurücksetzen, zu beschreiben oder zu lesen.

Mit der Erzeugung eines Paketes aus dem Schema input_output wird außerdem der integer Typ file_index vereinbart (Abkürzung FI).

type file_index is range 0 .. w;

Syntax:

Aus

ADA 22: I TYP

Semantik:

file_index ist ein implementierungsabhängiger integer Typ, der zur Angabe von Positionen innerhalb eines Files vom Komponententyp *T* dient.

Hinweise:

Die obere Grenze w ist implementierungsabhängig und wird durch file_index'last angegeben. Dieser Wert kann für Files verschiedener Komponententypen verschieden sein.

Man kann sich ein File auch als ein beliebig langes eindimensionales Array vorstellen. Die Elemente dieses Files lassen sich jedoch nicht wie bei einem Array einfach durch Indizierung bestimmen, sondern können nur über Positionsangaben und Ein/Ausgabeprozeduren angesprochen werden.

Wenn man entsprechende Unterprogramme zur Verfügung stellt, kann man ein File auch auffassen als eine Folge von Arrays

mit dem Komponententyp des Files. Z.B. kann in einem File von reellen Zahlen jeweils ein Paar aufgefaßt werden als Real- und Imaginärteil einer komplexen Zahl.

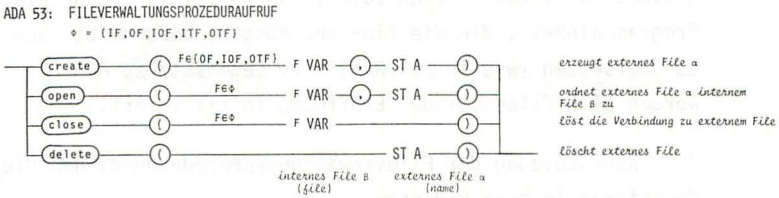
Ein File besitzt eine a k t u e l l e L ä n g e - die Anzahl der definierten Elemente -, eine E n d p o s i t i o n - die Nummer des letzten definierten Elementes, bzw. 0, falls das File leer ist. Ein Eingabefile besitzt eine a k t u e l l e L e s e p o s i t i o n - die Nummer des nächsten zu lesenden Elementes. Ein Ausgabefile besitzt eine a k t u e l l e S c h r e i b p o s i t i o n - die Nummer des Platzes, der als nächstes beschrieben wird. Ein Ein/Ausgabefile besitzt sowohl eine aktuelle Lese- als auch Schreibposition. Alle diese Positionsnummern sind vom implementierungsabhängigen integer Typ file_index.

Sollen die Files eines Programmes über das Ende des Programmes hinaus existieren, so müssen sie externen Files zugeordnet werden. Dazu dienen die FILEVERWALTUNGSPROZEDUREN.

Syntax:

Aus

ADA 53: FILEVERWALTUNGSPROZEDURAUFRUF



Wie für alle Fileverwaltungsprozeduren und Ein/Ausgabeunterprogramme ist der Name des formalen Argumentes für das interne File file und der für das externe File name. Der Name des externen Files ist implementierungsabhängig.

Diese 4 Prozeduren gelten auch für die im nächsten Abschnitt eingeführten Textfiles.

Semantik:

create schafft ein neues externes File α, dessen Elemente alle undefiniert sind. Dieses File wird eröffnet, d.h. dem internen File β zugeordnet. Die Schreib- oder Leseposition wird auf 1 gesetzt.

open ordnet das existierende externe File α dem internen File β zu. Die Schreibposition wird auf 1 gesetzt, die Leseposition auf das 1. definierte Element oder auf 1, falls das File leer ist.

close löst die Zuordnung zu einem externen File wieder auf.

delete löscht ein externes File α. Dieses darf nicht mehr eröffnet werden.

Einem internen File kann höchstens ein externes File zugeordnet werden. Jedoch können mehrere interne Files demselben externen File zugeordnet werden.

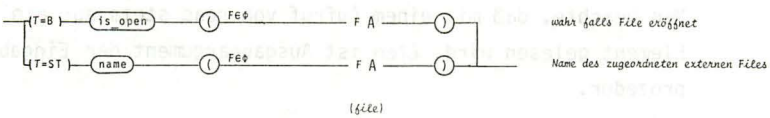
Diese vier Prozeduren regeln also die Zuordnung von externen zu internen Files. Im Gegensatz zu PASCAL geschieht das innerhalb des Programmes und nicht über die Programmparameterliste. Das heißt, daß die Ausführung des gleichen Programmes mit verschiedenen Dateien unmöglich ist, ohne die Programmeinheit, die die Ein- und Ausgabe beschreibt, neu zu übersetzen (wie in FORTRAN). Im Gegensatz zu PASCAL werden alle Files bei der Eröffnung initialisiert.

Zur Unterstützung der Fileverwaltungsprozeduren dienen die Funktionen is_open und name.

Syntax:

Aus

ADA 55: τ E/A STANDARDFUNKTIONSAUFRUF



Semantik:

is_open liefert den Wert true, falls dem File β ein externes File α zugeordnet ist, name bestimmt dessen Namen.

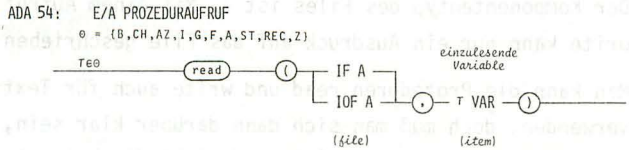
11.1.2 EIN- BZW. AUSGABE MIT FILES

Das Lesen eines Elementes von einem File vom Typ in_file oder inout_file geschieht mit der Standardprozedur read.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

read liest das Element an der aktuellen Leseposition und setzt diese auf das nächste definierte Element; falls keines vorhanden, wird um 1 erhöht. Die Ausnahme end_error wird ausgelöst, falls die Leseposition größer als die End-

position wird, data_error, falls der Wert nicht definiert oder (evtl.) nicht von passendem Typ ist. *T* bezeichnet den Komponententyp des Files.

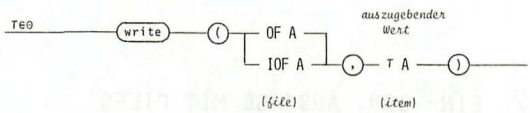
Man beachte, daß mit einem Aufruf von read stets nur ein Element gelesen wird. *item* ist Ausgabeargument der Eingabeprozedur.

Das Schreiben eines Wertes auf ein File vom Typ out_file oder inout_file geschieht mit der Standardprozedur write.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

write beschreibt den Platz der aktuellen Schreibposition und erhöht diese um 1. Falls die Endposition überschritten wird, so wird diese auch erhöht.

Der Komponententyp des Files ist *T*. Mit einem Aufruf von write kann nur ein Ausdruck auf das File geschrieben werden.

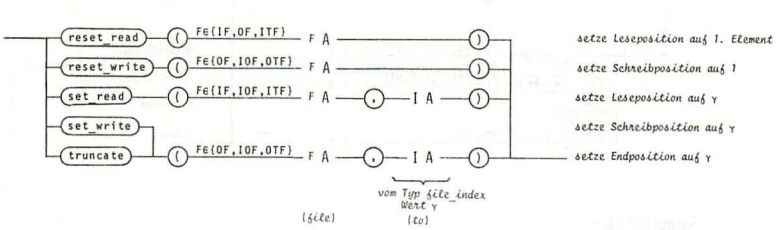
Man kann die Prozeduren read und write auch für Textfiles verwenden, doch muß man sich dann darüber klar sein, daß der Komponententyp character ist, und nicht die bei get und put übliche Konvertierung vorgenommen wird.

Normalerweise wird ein File von Anfang an Komponente für Komponente beschrieben und so auch wieder gelesen werden. Die folgenden Prozeduren verändern die Lese- bzw. Schreibposition.

Syntax:

Aus

ADA 53: FILEVERWALTUNGSPROZEDURAUFRUF



Semantik:

set_read bzw. set_write setzen die Lese- bzw. Schreibposition auf γ, reset_read setzt die Leseposition auf das erste definierte Element bzw. 1, falls kein Element definiert ist. reset_write setzt die Schreibposition auf 1. truncate setzt die Endposition auf den angegebenen Wert und ändert die Länge entsprechend. Ist der angegebene Wert größer als die Endposition, so erfolgt use_error.

Bemerkungen:

(1) Ein open auf ein leeres Eingabefile oder Ein/Ausgabefile bewirkt, daß die aktuelle Leseposition = 1 und die aktuelle Endposition = Ø ist. Auch durch set_read kann die Leseposition größer als die Endposition werden.

Die Ausnahme end_error wird nicht ausgelöst.

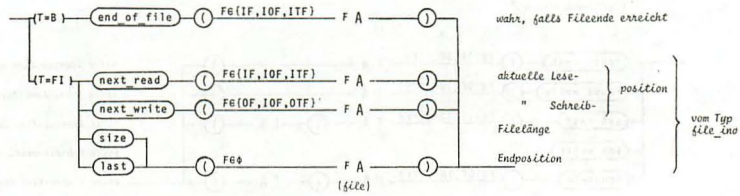
(2) Anders als in PASCAL bewirkt reset_write nicht, daß alle Elemente des Files undefiniert werden. Dem rewrite(f) von PASCAL entspricht in ADA
reset_write(f); truncate(f,1);

Zum Feststellen der aktuellen Position und Zustand der Files dienen die folgenden Funktionen.

Syntax:

Aus

ADA 55: T E/A STANDARDFUNKTIONSAUFRUF



Semantik:

Die Funktionen next_read bzw. next_write liefern die aktuelle Lese- bzw. Schreibposition, die Funktionen size bzw. last die Länge bzw. Endposition. end_of_file ist true, falls die Leseposition größer als die Endposition ist.

Es spielt keine Rolle, ob die externen Files sequentiell, indexsequentiell oder andersartig organisiert sind. Die Verwendung der verschiedenen Positionen erlaubt im Programm ein Hin- und Herspringen in Files. Sollte dieses bei speziellen externen Files verboten sein (z.B. Rücksetzen eines Lochstreifens), so erfolgt use_error.

11.1.3 AUSNAHMESITUATIONEN

Die Ausnahme status_error wird ausgelöst, falls

- ein create- oder open-Aufruf für eine bereits eröffnete Datei erfolgt (ADA 53).
- ein Aufruf einer anderen Prozedur oder Funktion für eine nicht eröffnete Datei erfolgt (ADA 53).

Die Ausnahme name_error wird ausgelöst, falls

- beim create-Aufruf eine externe Datei mit dem angegebenen Namen α bereits existiert (ADA 53).

- beim open-Aufruf eine externe Datei mit dem angegebenen Namen α nicht existiert oder der Zugriff auf die Datei α nicht erlaubt ist (ADA 53).

Die Ausnahme device_error wird ausgelöst, falls
- eine Ein/Ausgabeaktion, bedingt durch einen Systemfehler der betreffenden Datei oder des Gerätes, nicht korrekt beendet werden kann.

Die Ausnahme use_error wird ausgelöst, falls
- eine Aktion verlangt wird, die nicht mit dem externen File verträglich ist, z.B. das Rücksetzen eines Lochkartenlesers, das Schreiben auf eine schreibgeschützte Datei oder das Verlängern einer Datei vorgegebener Länge.

Die Ausnahme data_error wird ausgelöst, falls
- ein nicht definierter Wert gelesen wird.

Die Ausnahme data_error kann ausgelöst werden, falls
- ein Wert gelesen wird, der nicht mit dem Komponententyp kompatibel ist.

Die Ausnahme end_error wird ausgelöst, falls
- gelesen wird, obwohl end_of_file = true ist, d.h. die Leseposition größer als die Endposition ist.

11.2 DAS PAKET TEXT_IO

Die Ein/Ausgabemöglichkeiten, die das Paketschema input_output anbietet, reichen zwar zum Zwischenspeichern von Daten aus, doch für die Kommunikation mit einem menschlichen Benutzer müssen die Daten in einer lesbaren Form ein- und ausgegeben werden können. Zu dem Ein/Ausgabevorgang kommt jetzt also noch eine Konvertierung hinzu.

11.2.1 DIE VERWALTUNG VON TEXTFILES

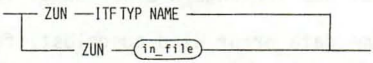
Das Paket text_io enthält eine Ausprägung des Schemas input_output mit dem Typ character und die Vereinbarung abgeleiteter Filetypen für Eingabe- und Ausgabefiles.

```
package character_io is new input_output (character);
   type in_file is new character_io.in_file;
   type out_file is new character_io.out_file;
```

Syntax:

ADA 22: ITF TYP

Eingabetextfiletyp



ADA 22: OTF TYP

Ausgabetextfiletyp

Semantik:

Es gibt nur Eingabedateien (Abkürzung ITF) und Ausgabedateien (OTF). Da mit Hilfe dieses Paketes für solche Files vom Komponententyp character zusätzliche Operationen möglich sind, werden diese Files jetzt Textfiles genannt. Als Zuname wird in der Regel der Name text_io auftreten.

Auf Textfiles sind selbstverständlich auch die Funktionen und Prozeduren des Paketes character_io anwendbar, jedoch sind nicht alle Kombinationen sinnvoll und haben klar definierte Wirkungen.

Die Zeichen, mit denen Textfiles beschrieben oder gelesen werden, müssen zu den 95 druckbaren ASCII-Zeichen gehören. Die Wirkung eines Programmes, das dagegen verstößt, ist im-

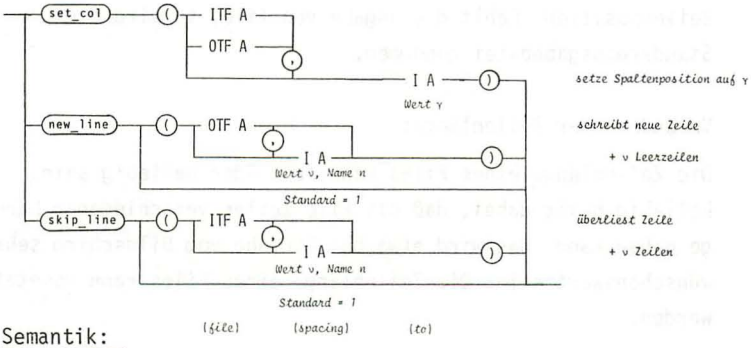
plementierungsabhängig.

Textfiles bestehen aus einer Folge von Zeilen, jede Zeile aus aufeinanderfolgenden Zeichenpositionen, den Spalten. Die Nummerierung beginnt jeweils bei 1. Jedes Textfile besitzt also eine Zeilen- und eine Spaltenposition, die beide von dem integer Untertyp natural (Abkürzung N) sind. Die Zeilen- und Spaltenposition beschreiben die für den nächsten Ein- oder Ausgabebefehl relevante Position, sie können durch folgende Prozeduraufrufe gesetzt werden.

Syntax:

Aus

ADA 53: FILEVERWALTUNGSPROZEDURAUFRUF



Semantik:

set_col setzt die Spaltenposition auf den Wert γ. Falls γ größer als die Zeilenlänge ist, erfolgt layout_error.

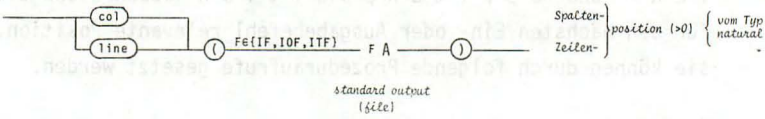
new_line beendet die Ausgabe, skip_line die Eingabe einer Zeile, es werden ν-1 Leerzeilen ausgegeben bzw. ν-1 Zeilen überlesen. Alle Prozeduren sind auch in einer Version vorhanden, in der auf die Angabe des Textfiles verzichtet werden kann. set_col und new_line beziehen sich dann auf die aktuelle Standardausgabedatei, skip_line auf die aktuelle Standardeingabedatei. Das formale Argument n (an anderer Stelle steht spacing) hat als Vorbesetzung den Wert 1.

Zur Ermittlung der aktuellen Zeilen- bzw. Spaltenposition dienen die Funktionen col und line.

Syntax:

Aus

ADA 55: I E/A STANDARDFUNKTIONSAUFRUF



Semantik:

col liefert die aktuelle Spaltenposition, line die aktuelle Zeilenposition. Fehlt die Angabe von file, so wird die Standardausgabedatei genommen.

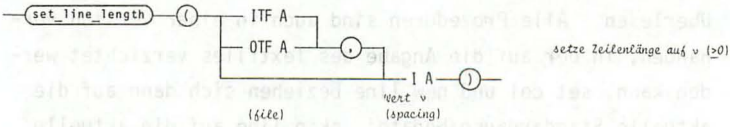
Verändern der Zeilenlänge:

Die Zeilenlänge eines Files kann fest oder beliebig sein. Beliebig heißt dabei, daß das File Zeilen verschiedener Länge haben kann. Das wird etwa bei Eingabe vom Bildschirm sehr wünschenswert sein. Die Zeilenlänge eines Files kann gesetzt werden.

Syntax:

Aus

ADA 53: FILEVERWALTUNGSPROZEDURAUFRUF



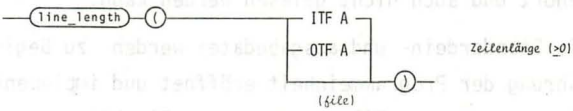
Semantik:

Die Zeilenlänge des angegebenen files oder der Standardausgabedatei wird auf ν gesetzt. Ist ν = 0, so ist die Zeilenlänge beliebig.

Syntax:

Aus

ADA 55: I E/A STANDARDFUNKTIONSAUFRUF



Semantik:

Der Wert der Funktion ist die Zeilenlänge des Textfiles. Wieder bedeutet der Wert 0, daß die Zeilenlänge des Textfiles beliebig ist.

Fehlt die Angabe des Files, so wird die Standardausgabedatei verwendet.

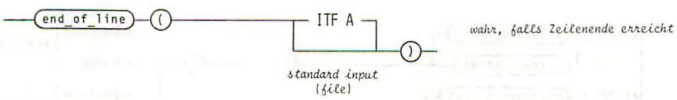
Ende einer Zeile:

Für Textfiles mit beliebiger Länge kann das Ende einer Zeile einer Eingabedatei mit der Funktion end_of_line festgestellt werden.

Syntax:

Aus

ADA 55: B E/A STANDARDFUNKTIONSAUFRUF



Semantik:

Falls die Zeilenlänge in dem angegebenen Textfile (oder der Standardeingabedatei) beliebig ist und die aktuelle Spaltenposition die tatsächliche Länge der aktuellen Zeile überschreitet, so ist der Wert von end_of_line = true, ansonsten = false.

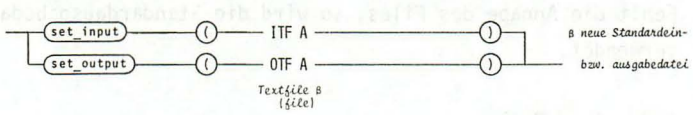
Das Ende einer Zeile eines Textfiles wird durch ein spezielles Zeilenendezeichen markiert, das nicht zum Zeichensatz gehört und auch nicht gelesen werden kann.

Die Standardein- und ausgabedatei werden zu Beginn der Ausführung der Programmeinheit eröffnet und implementierungsbedingten externen Files zugeordnet. Sie können vom Benutzer festgestellt und umdefiniert werden.

Syntax:

Aus

ADA 53: FILEVERWALTUNGSPROZEDURAUFRUF



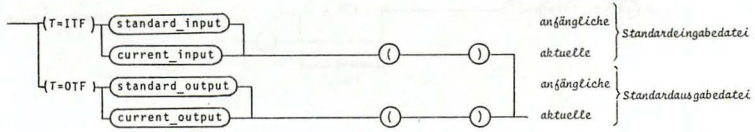
Semantik:

Das angegebene Textfile wird die aktuelle Standardein- bzw. ausgabedatei.

Syntax:

Aus

ADA 55: T E/A STANDARDFUNKTIONSAUFRUF



Semantik:

Die Funktionen standard_input bzw. standard_output liefern die anfängliche Standardein- bzw. Standardausgabedatei, current_input bzw. current_output die aktuelle.

Hinweise:

Man kann sich ein Textfile wie eine Matrix vorstellen, deren Zeilen verschieden lang sein können.

Files mit fester Zeilenlänge eignen sich gut für formatierte Ein/Ausgabe.

11.2.2 EIN- BZW. AUSGABE MIT TEXTFILES

Die Prozeduren get bzw. put sind mehrfach überladene Prozeduren für die Eingabe bzw. Ausgabe von Daten verschiedener Typen mit einem Textfile.

11.2.2.1 EIN- BZW. AUSGABE FÜR ZEICHEN UND ZEICHENKETTEN

Das Paket text_io enthält die Paketerzeugung des Paketes character_io.

```
package character_io is new input_output (character);
```

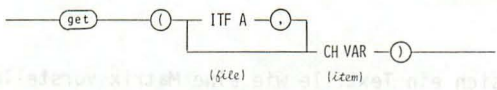
Die Textfiletypen in_file und out_file werden vereinbart und hier mit ITF und OTF abgekürzt.

Für das Einlesen von Zeichen steht die Prozedur get bereit.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

get liest das Zeichen des angegebenen files oder der Standardeingabedatei, das durch die aktuelle Zeilen- und Spaltenposition bestimmt ist. Die Spaltenposition wird um 1 erhöht. Ist dann bei einem File fester Länge die Spaltenposition größer als die Zeilenlänge, so wird die Zeilenposition um 1 erhöht und die Spaltenposition auf 1 gesetzt, also an den Anfang der nächsten Zeile gesprungen. Das gelesene Zeichen wird in der CH Variablen *item* übergeben.

Hinweis:

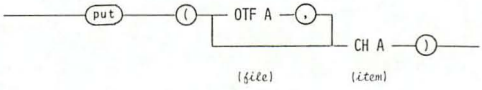
Das Zeichen auf dem Textfile entspricht nicht der Syntax einer CH KONSTANTEN, denn es wird nicht in Apostrophe eingeschlossen.

Zur Ausgabe eines Wertes vom Typ character dient die Prozedur put.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

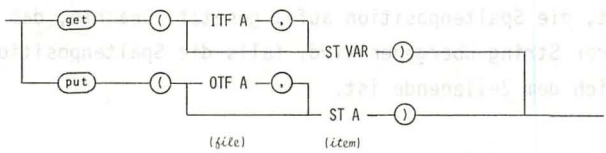
Der CH Ausdruck *item* wird ausgewertet und an die Stelle des angegebenen *file* bzw. der Standardausgabedatei geschrieben, die durch die aktuelle Zeilen- und Spaltenposition bestimmt wird. Die Spaltenposition wird um 1 erhöht. Ist dann bei einem File fester Länge die Spaltenposition größer als die Zeilenlänge, so wird ein Zeilenendezeichen ausgegeben und die Zeilenposition um 1 erhöht und die Spaltenposition auf 1 gesetzt.

Man kann ein Textfile, das ja eine Folge von Zeichen beschreibt, auch als eine Folge von Zeichenketten auffassen. Die Prozeduren get bzw. put für Zeichenketten bedeuten die mehrfache, der Stringlänge entsprechende Ausführung von get bzw. put für Zeichen.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

```
get (item)        ist äquivalent zu
   for i in item'first .. item'last
   loop
      get (item(i));        -- get für CH Variable
   end loop;
```

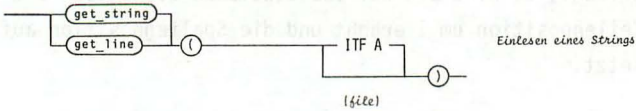
Es werden also der Stringlänge entsprechend viele Zeichen gelesen.

Für die Eingabe existieren zusätzlich die Funktionen get_string und get_line.

Syntax:

Aus

ADA 55: ST E/A STANDARDFUNKTIONSAUFRUF



Semantik:

get_string überliest auftretende Trennzeichen, das sind Leerzeichen, Tabulatoren oder Zeilenendezeichen, und übergibt den folgenden String als Ergebnis. Der String ist von beliebiger Länge und wird durch ein Trennzeichen beendet.

get_line übergibt alle Zeichen von der aktuellen Leseposition (bestimmt durch Zeilen- und Spaltenposition) bis zum Zeilenende als Ergebnis. Die Zeilenposition wird um 1 erhöht, die Spaltenposition auf 1 gesetzt. Beachte, daß ein leerer String übergeben wird, falls die Spaltenposition gleich dem Zeilenende ist.

```
item:= get_string;
```

ist äquivalent zu

```
loop            -- Überlesen von Trennzeichen
   while end_of_line ()
   loop skip_line;
   end loop;
   get (ch);    -- Bei fester Zeilenlänge wird Zeilenende-
                -- zeichen überlesen.
```

```
   exit when ch /= '' or else ch /= ascii.ht;
end loop;
                    -- ch ist 1. Zeichen des String
i:= 1;
loop
   item (i):= ch;
   get (ch);
   exit when ch = '' or else ch = ascii.ht
           or else col() = 1     -- Zeilenende bei fester
                                 -- Laenge
           or else end_of_line ();   -- Zeilenende bei
                                     -- variabler Laenge
```

```
item:= get_line(f);
```

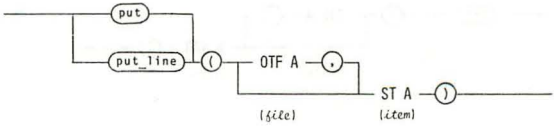
ist äquivalent zu

```
i:= 1;
while col(f) <= line_length(f)
loop
   get (f,item(i));
   i:= i+1;
end loop;
if end_of_line (f) then skip_line (f);
   -- Bei fester Laenge automatisch mit get.
```

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

Der ST Ausdruck *item* wird ausgewertet. Fehlt die Angabe von *file* , so wird auf die Standardausgabedatei geschrieben.

put schreibt der Stringlänge entsprechend viele Zeichen. put_line fügt zusätzlich ein Zeilenendezeichen an und erhöht die Zeilenposition um 1 und setzt die Spaltenposition auf 1.

```
put (f,item);
```

ist äquivalent zu

```
for i in item'first .. item'last
loop
   put (f,item(i));    -- put für CH Ausdruck
end loop;
```

```
put_line (f,item);
```

ist äquivalent zu

```
put (f,item);
new_line (1);
```

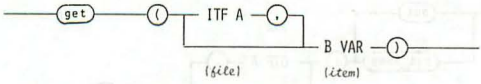
11.2.2.2 EIN- BZW. AUSGABE FÜR LOGISCHE WERTE

Die Prozeduren get und put für logische Werte funktionieren wie folgt.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

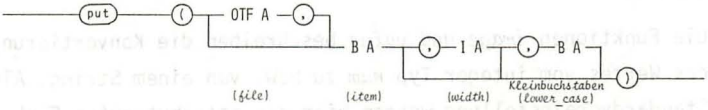
Vom angegebenen File oder der Standardeingabedatei wird ein Name gelesen, dabei wird nicht zwischen Groß- und Kleinbuchstaben unterschieden. Ist der Name gleich true oder false, so wird der entsprechende Wert in *item* übergeben, sonst erfolgt data_error.

Zeilenendezeichen, Leerzeichen und Tabulatoren werden am Anfang überlesen. Die Eingabe endet beim 1. Zeichen, das nicht der Syntax eines Namens (ADA 65) entspricht.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

Der dem Wert von *item* entsprechende Name (true oder false) wird auf ein Ausgabefeld der Breite *width* - mindestens jedoch der Länge des Namens - linksbündig geschrieben. Normalerweise in Großbuchstaben; falls *lower_case* true ist, jedoch in Kleinbuchstaben.

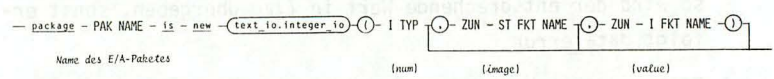
11.2.2.3 EIN- BZW. AUSGABE FÜR GANZZAHLIGE WERTE

Die Ein- und Ausgabeprozeduren stehen nicht direkt zur Verfügung, sie sind in dem Paketschema integer_io zusammengefaßt. Für jeden integer Typ muß aus diesem Schema ein Ein-Ausgabepaket erzeugt werden.

Syntax:

Aus

ADA 11: PAKETERZEUGUNG



Semantik:

Es wird ein Ein-Ausgabepaket für den angegebenen I Typ *num* erzeugt.

Der Zuname text_io kann entfallen, falls die Größen des Paketes text_io durch eine use-Klausel direkt sichtbar gemacht wurden.

Die Funktionen *image* und *value* beschreiben die Konvertierung des Wertes vom integer Typ *num* zu bzw. von einem String. Als Standardvoreinstellung werden hier die attributierten Funktionen num'image und num'value eingesetzt.

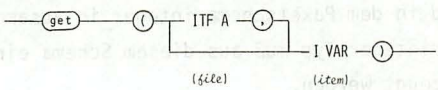
Bei Angabe der Schemaargumente dem Namen nach (num, image und value sind die Namen der formalen Schemaargumente) kann sich selbstverständlich die Reihenfolge ändern.

Eine so erzeugte Ausprägung des Schemas integer_io stellt die Prozeduren get und put für Werte des integer Typs num zur Verfügung.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

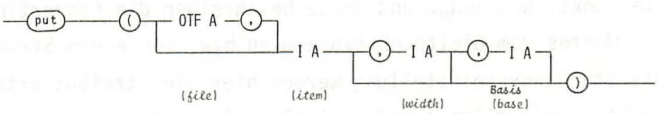
Es wird eine der Syntax entsprechende, gegebenenfalls mit einem Vorzeichen versehene integer Literalkonstante gelesen, mit Hilfe der Funktion value konvertiert und dem Ausgabeargument *item* zugewiesen.

Führende Leerzeichen, Tabulatoren und Zeilenendezeichen werden überlesen. Die Eingabe endet beim 1. Zeichen, das nicht der Syntax einer integer Literalkonstanten (ADA 39) entspricht.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

put gibt mit der Feldbreite *width* (jedoch mindestens alle benötigten Ziffern) rechtsbündig den Wert des Ausdruckes *item* in Form einer integer Konstanten ohne Unterstriche und Exponententeil aus. Falls eine Basis ≠ 10 angegeben wird, so erfolgt die Ausgabe entsprechend.

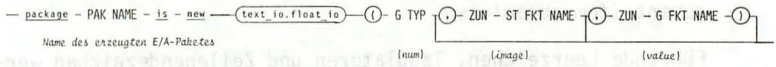
11.2.2.4 EIN- BZW. AUSGABE FÜR GLEITPUNKTZAHLEN

Die Ein- und Ausgabeprozeduren stehen nicht direkt zur Verfügung, sie sind in dem Paketschema float_io zusammengefaßt. Für jeden Gleitpunkttyp muß aus diesem Schema ein Ein-Ausgabepaket erzeugt werden.

Syntax:

Aus

ADA 11: PAKETERZEUGUNG



Semantik:

Es wird ein Ein/Ausgabepaket für den angegebenen G Typ erzeugt.

Der Zuname text_io kann entfallen, falls die Größen des Paketes text_io durch eine use-Klausel direkt sichtbar gemacht werden.

Die Funktionen *image* und *value* beschreiben die Konvertierung des Wertes vom Gleitpunkttyp *num* zu bzw. von einem String. Als Standardvoreinstellung werden hier die attributierten Funktionen num'image und num'value eingesetzt.

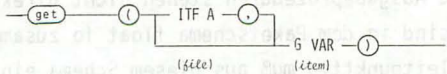
Bei Angabe der Schemaargumente dem Namen nach (num, image und value sind die Namen der formalen Schemaargumente) kann sich selbstverständlich die Reihenfolge ändern.

Eine so erzeugte Ausprägung des Schemas float_io stellt die Prozeduren get und put für Werte des Gleitpunkttyps num zur Verfügung.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

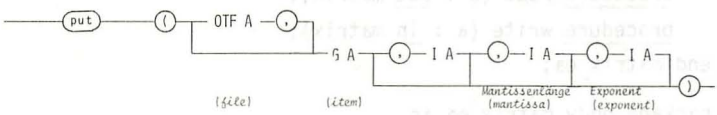
Es wird eine der Syntax entsprechende, gegebenenfalls mit einem Vorzeichen versehene reelle Literalkonstante gelesen, mit Hilfe der Funktion value konvertiert und dem Ausgabeargument *item* zugewiesen.

Führende Leerzeichen, Tabulatoren und Zeilenendezeichen werden überlesen. Die Eingabe endet beim 1. Zeichen, das nicht der Syntax einer reellen Literalkonstanten (ADA 40) entspricht.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

Der Wert des Gleitpunktausdruckes *item* wird mit *width* Druckpositionen rechtsbündig ausgegeben: Vorzeichen (falls negativ), 1. Mantissenstelle, Dezimalpunkt, *mantissa* - 1 Mantissenstellen, E, Vorzeichen des Exponenten, *exponent* Exponentenziffern.

Fehlt *mantissa*, so werden num'digits Mantissenstellen ausgegeben, ist mantissa < num'digits, so wird gerundet, ist mantissa > num'digits, so werden führende Nullen vor der 1. Mantissenziffer ausgegeben. (num'digits bezeichnet die in der Typvereinbarung des Gleitpunkttyps, für den das E/A-Paket erzeugt wurde, angegebene Anzahl der dezimalen Mantissenstellen).

Fehlt *exponent*, so werden 2 Exponentenziffern ausgegeben. Es

werden mindestens so viele Exponentenziffern ausgegeben, wie gebraucht werden. Gegebenenfalls werden führende Nullen eingefügt, um die angegebene Ziffernzahl zu erreichen.

Eine Konvertierung zu anderen Basen ist nicht vorgesehen.

Beispiel: Schema zur Ein- und Ausgabe von quadratischen Gleitpunktmatrizen beliebiger Dimension

```
with text_io;
generic
   type indextyp is (<>);
   type floattyp is digits <>;
   type matrix is array (indextyp,indextyp) of floattyp;

package matrix_ea is
   procedure read (a : out matrix);
   procedure write (a : in matrix);
end matrix_ea;

package body matrix_ea is
   use text_io;
   package fio is new float_io (floattyp);
     use fio;
   procedure read (a : out matrix) is
   begin
     for i in indextyp
     loop
        for j in indextyp
        loop
           get (a(i,j));
        end loop;
     end loop;
   end read;

   procedure write (a : in matrix) is
   begin
     for i in indextyp
```

```
      loop
         for j in indextyp
         loop
            put (a(i,j));
         end loop;
         newline;
      end loop;
   end write;
end;
```

Ausprägung dieses Schemas für 2×2-Matrizen vom Typ float

```
type matrix is array (integer range 1 .. 2, integer range
                      1 .. 2) of float;
package mat2_2_ea is new matrix_ea (integer range 1 .. 2,
        float, matrix);
```

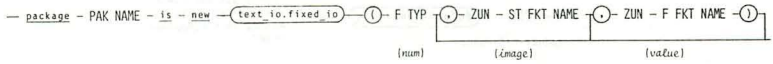
11.2.2.5 EIN- BZW. AUSGABE FÜR FESTPUNKTZAHLEN

Die Ein- und Ausgabeprozeduren stehen nicht direkt zur Verfügung, sie sind in dem Paketschema fixed_io zusammengefaßt. Für jeden Festpunkttyp muß aus diesem Schema ein Ein-Ausgabepaket erzeugt werden.

Syntax:

Aus

ADA 11: PAKETERZEUGUNG



Semantik:

Es wird ein Ein/Ausgabepaket für den angegebenen Festpunkttyp *num* erzeugt.

Der Zuname text_io kann entfallen, falls die Größen des Paketes text_io durch eine use-Klausel direkt sichtbar gemacht werden.

Die Funktionen *image* und *value* beschreiben die Konvertierung des Wertes vom Festpunkttyp *num* zu bzw. von einem Text. Als Standardvoreinstellung werden hier die attributierten Funktionen num'image und num'value eingesetzt.

Bei Angabe des Schemaargumentes dem Namen nach (num, image und value sind die Namen der formalen Schemaargumente) kann sich selbstverständlich die Reihenfolge ändern.

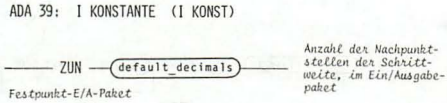
In dem erzeugten Ein-Ausgabepaket werden die beiden Konstanten delta_image und default_decimals vereinbart.

```
delta_image : constant string := image (num'delta -
                                  integer (num'delta));
default_decimals : constant integer := delta_image'length-2;
```

Syntax:

Aus

Aus



Semantik:

delta_image bezeichnet die Stringdarstellung des gebrochenen Anteils der Schrittweite delta des Festpunkttyps, für den

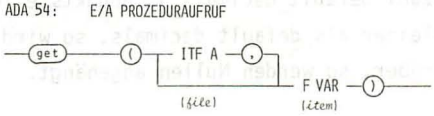
das E/A-Paket erzeugt wurde. default_decimals beschreibt die Anzahl der Nachpunktstellen dieser Schrittweite delta.

Eine Ausprägung des Schemas fixed_io stellt die Prozeduren get und put für Werte des Festpunkttyps num zur Verfügung.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

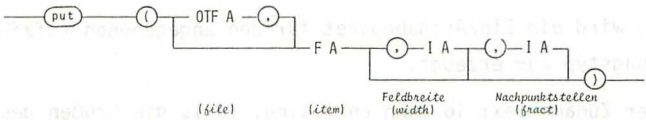
Es wird eine der Syntax entsprechende, gegebenenfalls mit einem Vorzeichen versehene reelle Literalkonstante gelesen, mit Hilfe der Funktion *value* konvertiert und dem Ausgabeargument *item* zugewiesen.

Führende Leerzeichen, Tabulatoren und Zeilenendezeichen werden überlesen. Die Eingabe endet beim 1. Zeichen, das nicht der Syntax einer reellen Literalkonstante (ADA 40) entspricht.

Syntax:

Aus

ADA 54: E/A PROZEDURAUFRUF



Semantik:

Der Wert des Festpunktausdruckes *item* wird mit *width* Ausgabepositionen rechtsbündig ausgegeben: Vorzeichen, falls negativ, ganzzahliger Anteil, Dezimalpunkt, *fract* Nachpunktstellen. Es wird mindestens eine Stelle des ganzzahligen Anteils ausgegeben. Falls erforderlich, werden führende Nullen eingefügt. Fehlt die Angabe von *fract*, so werden die aktuelle Stellenzahl default_decimals Nachpunktstellen ausgegeben. Ist *fract* kleiner als default_decimals, so wird gerundet, ist *fract* größer, so werden Nullen angehängt.

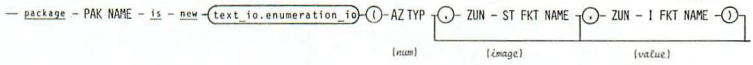
11.2.2.6 EIN- BZW. AUSGABE FÜR WERTE VON AUFZÄHLUNGSTYPEN

Die Ein- und Ausgabeprozduren stehen nicht direkt zur Verfügung, sie sind in dem Paketschema enumeration_io zusammengefaßt. Für jeden Aufzählungstyp muß aus diesem Schema ein Ein-Ausgabepaket erzeugt werden.

Syntax:

Aus

ADA 11: PAKETERZEUGUNG



Semantik:

Es wird ein Ein/Ausgabepaket für den angegebenen Aufzählungstyp *num* erzeugt.

Der Zuname text_io kann entfallen, falls die Größen des Paketes text_io durch eine use-Klausel direkt sichtbar gemacht werden.

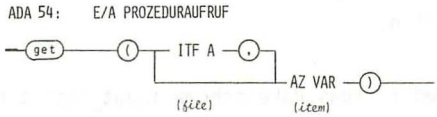
Die Funktionen *image* und *value* beschreiben die Konvertierung des Wertes vom Aufzählungstyp *num* zu bzw. von einem Text. Als Standardvoreinstellung werden hier die attributierten Funktionen num'image und num'value eingesetzt.

Bei Angabe des Schemaarguments dem Namen nach (num, image und value sind die Namen der formalen Schemaargumente) kann sich selbstverständlich die Reihenfolge ändern.

Eine so erzeugt Ausprägung des Schemas enumeration_io stellt die Prozeduren get und put für Werte des Aufzählungstyps num zur Verfügung.

Syntax:

Aus

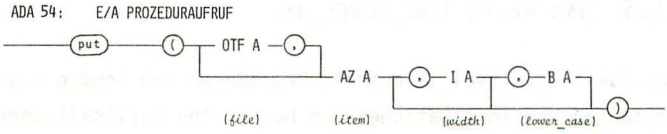


Semantik:

Es wird eine AZ Literalkonstante gelesen, d.h. entweder ein in Apostrophe eingeschlossenes Zeichen oder ein Name, dessen interner Wert mit der Funktion *value* ermittelt und dem Ausgabeargument *item* zugewiesen wird.

Syntax:

Aus



Semantik:

Der dem Wert des AZ Ausdrucks *item* entsprechende Name wird in ein Feld der Breite *width* linksbündig ausgegeben. Standardmäßig werden Großbuchstaben verwendet, falls *lower_case* den Wert true hat, jedoch Kleinbuchstaben.

Hinweise:

Für jeden numerischen Typ muß ein Ein-Ausgabepaket erzeugt werden.

get und put können jeweils nur für einen Wert eines Typs verwendet werden.

Ein/Ausgabe von Aufzählungstypen ist in ADA möglich.

Für strukturierte Typen lassen sich leicht Ein/Ausgabepakete zusammenstellen.

Die Ausnahmen für das Paketschema input_output haben auch für text_io Gültigkeit.

data_error wird ausgelöst, falls der zu lesende Wert nicht der verlangten Syntax entspricht.
Zusätzlich gibt es noch die Ausnahme layout_error, die ausgelöst wird, falls die Spaltenposition auf einen Wert gesetzt wird, der größer als die Endposition ist, oder falls bei einer Datei fester Zeilenlänge ein Zeilenendezeichen innerhalb einer Zeile gelesen wird.

11.3 DAS PAKET LOW_LEVEL_IO

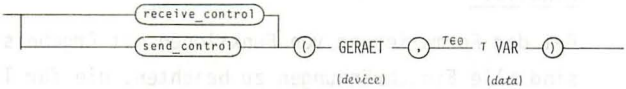
Das Paket low_level_io enthält Prozeduren zum Senden und Empfangen von Informationen von bestimmten physikalischen Geräten. Für die zur Verfügung stehenden Geräte ist implementierungsbedingt je eine Prozedur send_control und

receive_control anzulegen.

Syntax:

ADA 54: E/A PROZEDURAUFRUF

Aus



Semantik:

GERAET ist die Bezeichnung eines Objektes von einem implementierungsabhängigen Gerätetyp. Der Typ *T* des Ein/Ausgabeargumentes *data* hängt von dem entsprechenden Gerät ab.

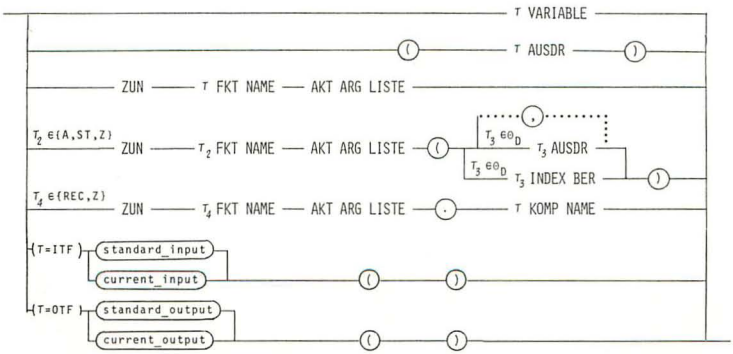
11.4 DER AUSDRUCK FÜR FILES

Filetypen sind limitierte, geschützte Typen. Es gibt keine Standardoperatoren. Man kann Files nicht vergleichen. Variable vom Filetyp wirken wie Konstante, da keine Zuweisung möglich ist. Jedoch können Files das Ergebnis von Funktionen und überladenen Operatoren sein (vgl. 3., 5.).

Syntax:

ADA 28: T AUSDRUCK (T AUSDR, T A) T ∈ Φ

Φ = {IF,OF,IOF,ITF,OTF}



Semantik:

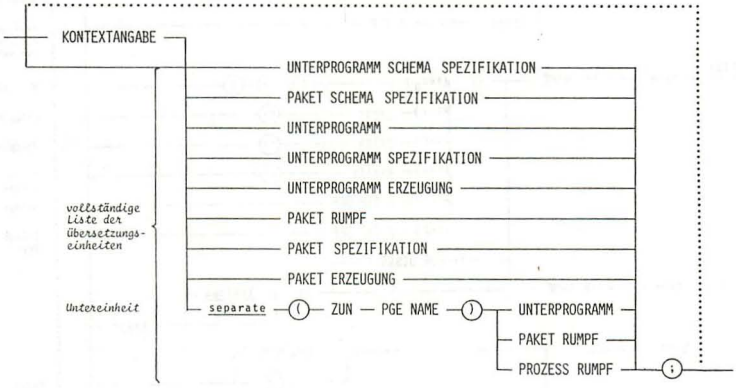
Das Ergebnis eines Ausdruckes ist ein File. Die Wirkung der Standarfunktionen wurde in Abschnitt 11.2.1 beschrieben.

Hinweise:

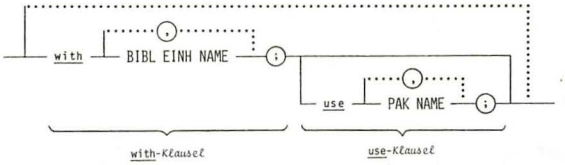
Bei der Formulierung von Funktionen mit Ergebnistyp file sind alle Einschränkungen zu beachten, die für limitierte, geschützte Typen gelten (s. 2.1.6).

C. SYNTAXDIAGRAMME

ADA 1: PROGRAMM



ADA 2: KONTEXTANGABE

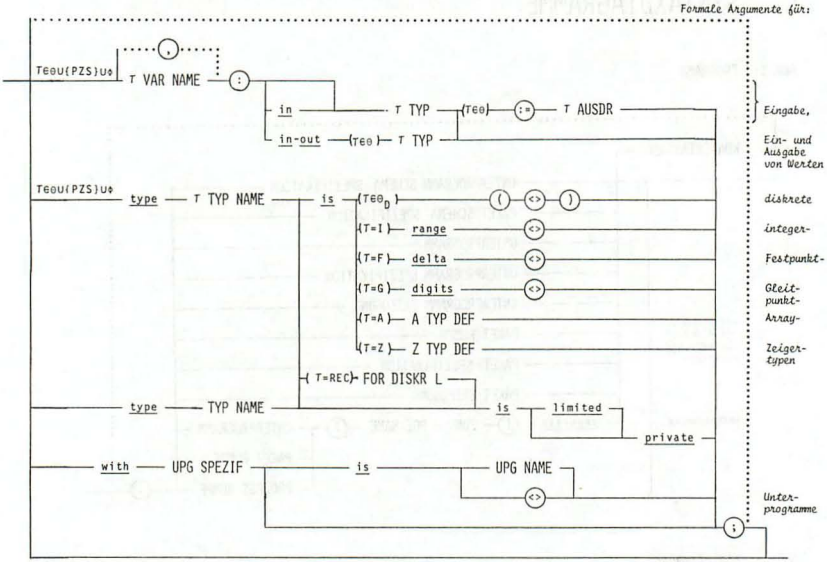


ADA 3: UNTERPROGRAMM SCHEMA SPEZIFIKATION

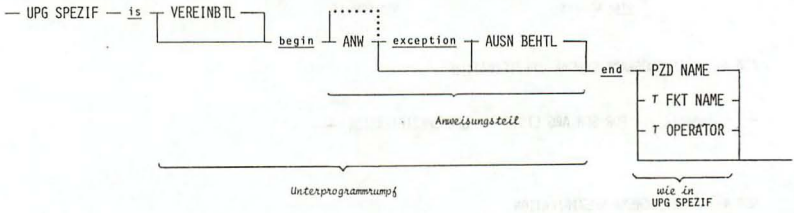
ADA 4: PAKET SCHEMA SPEZIFIKATION

ADA 5: FORMALE SCHEMA ARGUMENT LISTE (FOR SCH ARG LISTE)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Φ = {IF,OF,IOF,ITF,OTF} Θ_D = {B,CH,AZ,I}

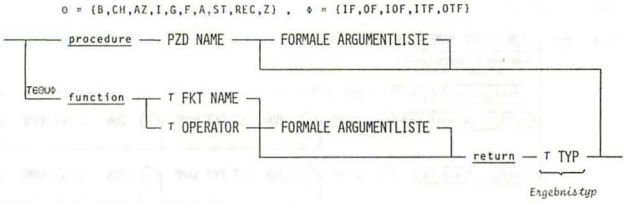


ADA 6: UNTERPROGRAMM (UPG)



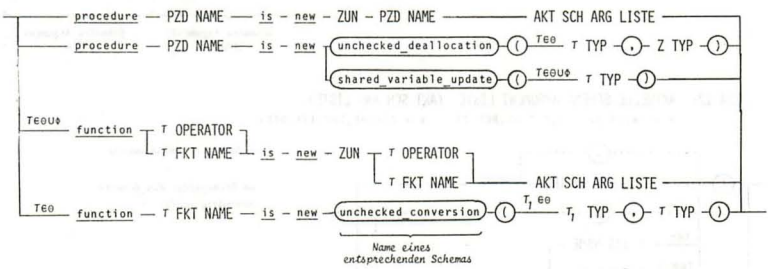
ADA 7: UNTERPROGRAMM SPEZIFIKATION (UPG SPEZIF)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}

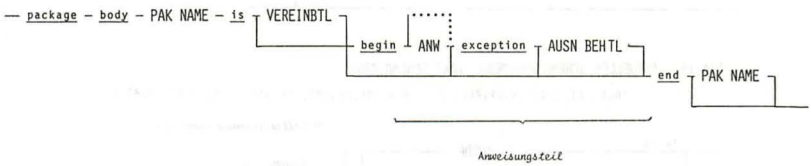


ADA 8: UNTERPROGRAMM ERZEUGUNG

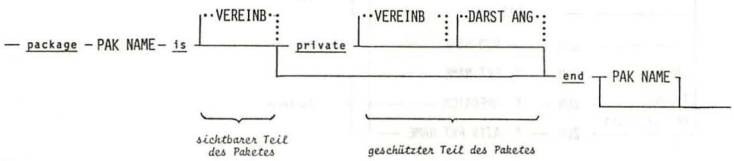
Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Φ = {IF,OF,IOF,ITF,OTF}



ADA 9: PAKETRUMPF

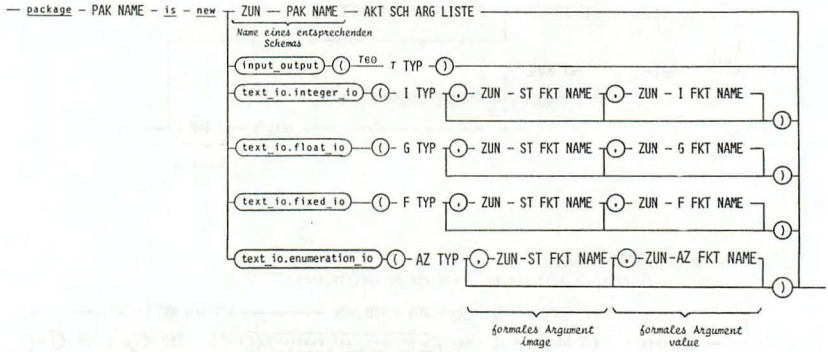


ADA 10: PAKETSPEZIFIKATION (PAK SPEZIF)



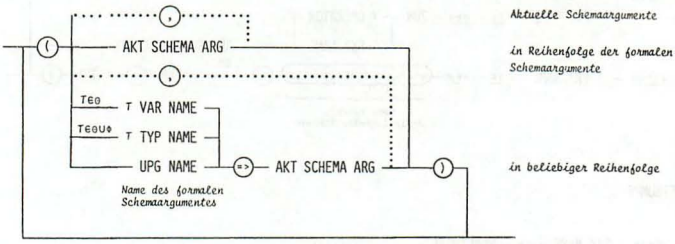
ADA 11: PAKETERZEUGUNG

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$



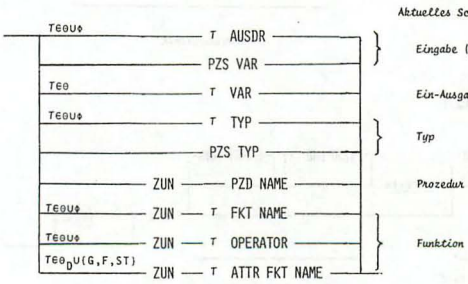
ADA 12: AKTUELLE SCHEMA ARGUMENT LISTE (AKT SCH ARG LISTE)

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$ $\Phi = \{IF,OF,IOF,ITF,OTF\}$

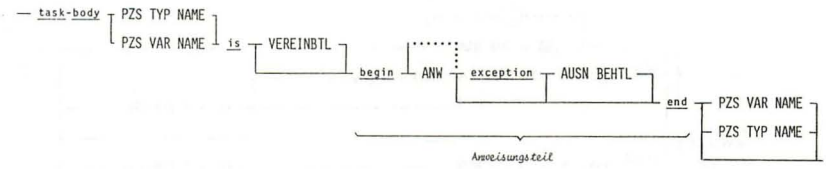


ADA 13: AKTUELLES SCHEMA ARGUMENT (AKT SCHEMA ARG)

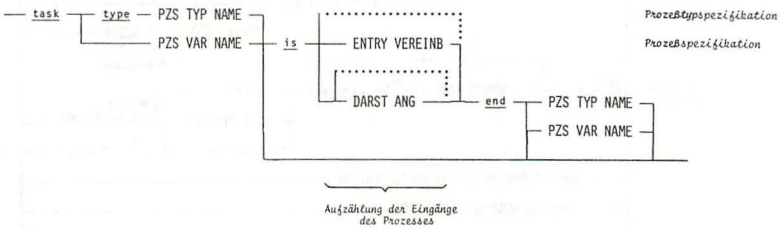
$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$ $\Phi = \{IF,OF,IOF,ITF,OTF\}$ $\Theta_D = \{B,CH,AZ,I\}$



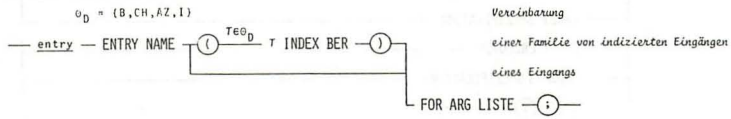
ADA 14: PROZESSRUMPF



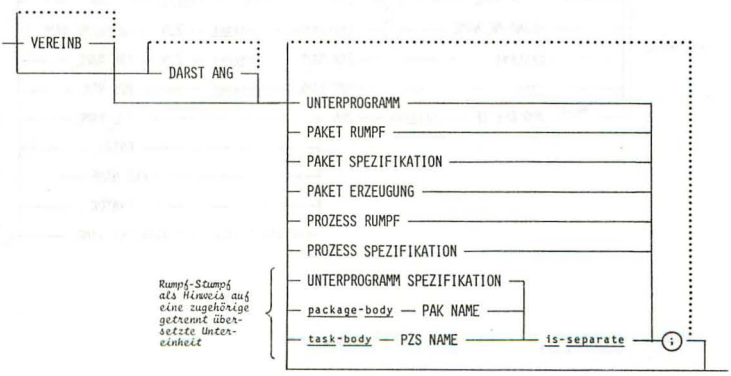
ADA 15: PROZESSSPEZIFIKATION



ADA 16: ENTRY VEREINBARUNG (ENTRY VEREINB)



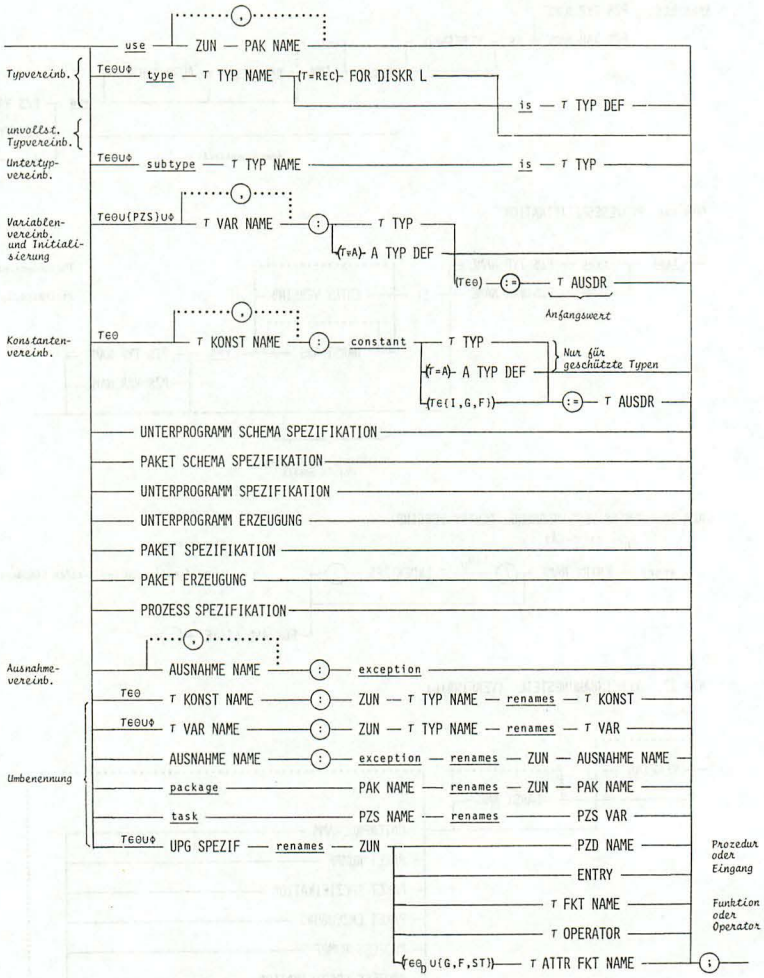
ADA 17: VEREINBARUNGSTEIL (VEREINBTL)



ADA 18: VEREINBARUNG (VEREINB)

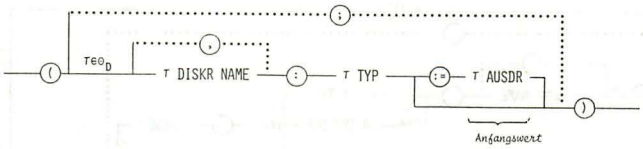
Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}

Θ_D = {B,CH,AZ,I}



ADA 19: FORMALE DISKRIMINANTENLISTE (FOR DISKR L)

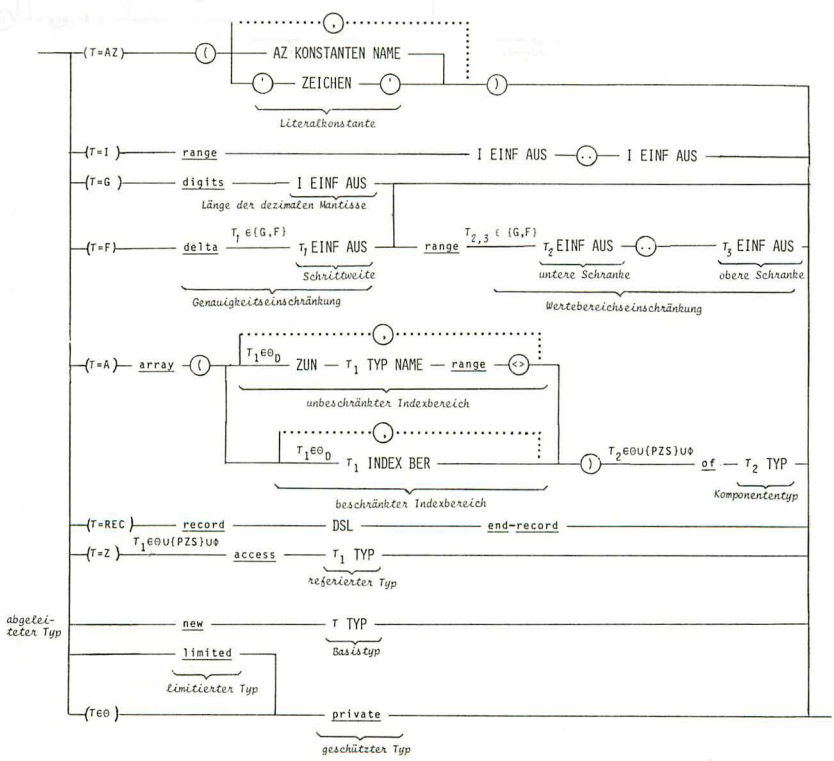
Θ_D = {B,CH,AZ,I}



ADA 20: T TYP DEFINITION (T TYP DEF) T ∈ Θ ∪ Φ

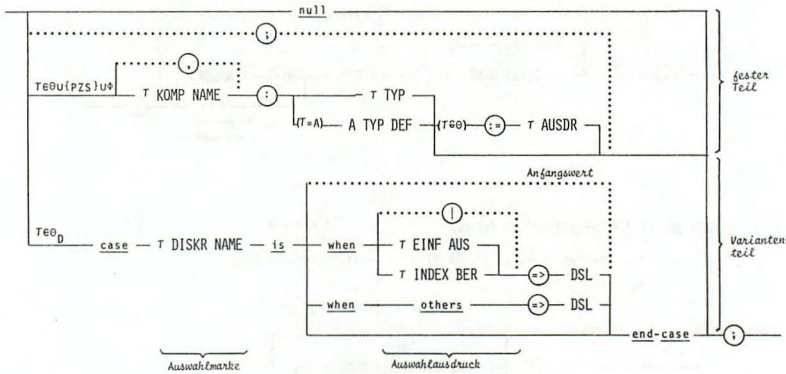
Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}

Θ_D = {B,CH,AZ,I}



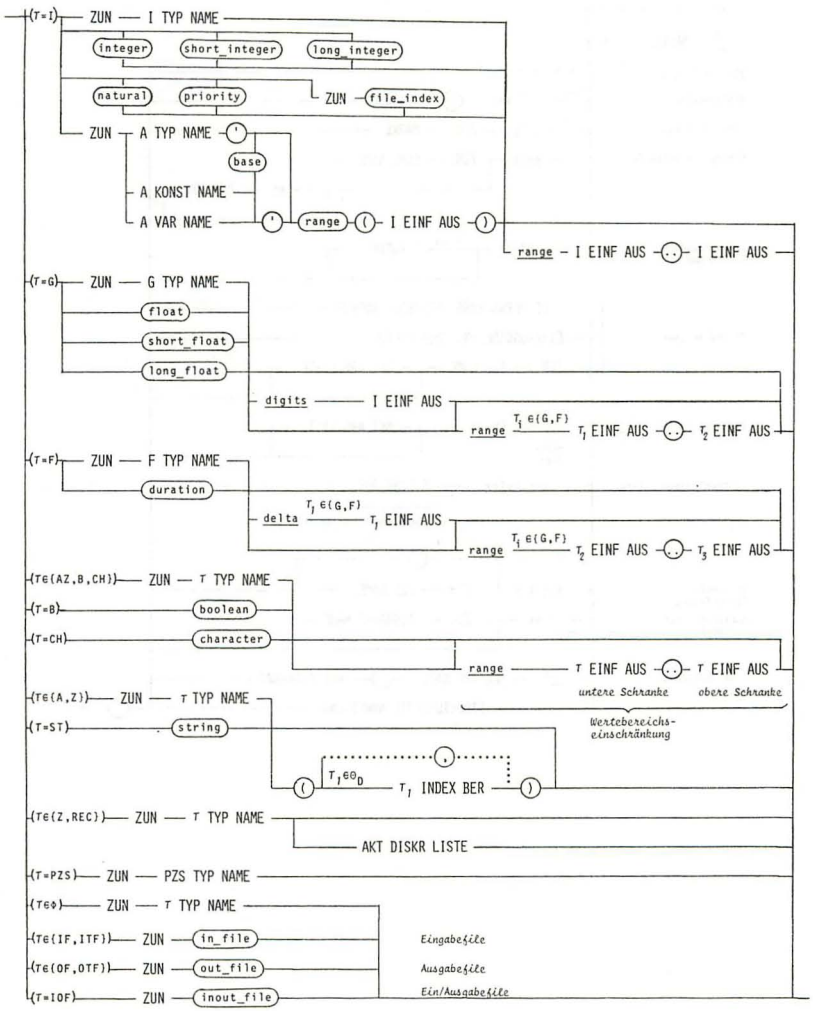
ADA 21: DATENSATZLISTE (DSL)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Φ = {IF,OF,IOF,ITF,OTF} Θ_D = {B,CH,AZ,I}



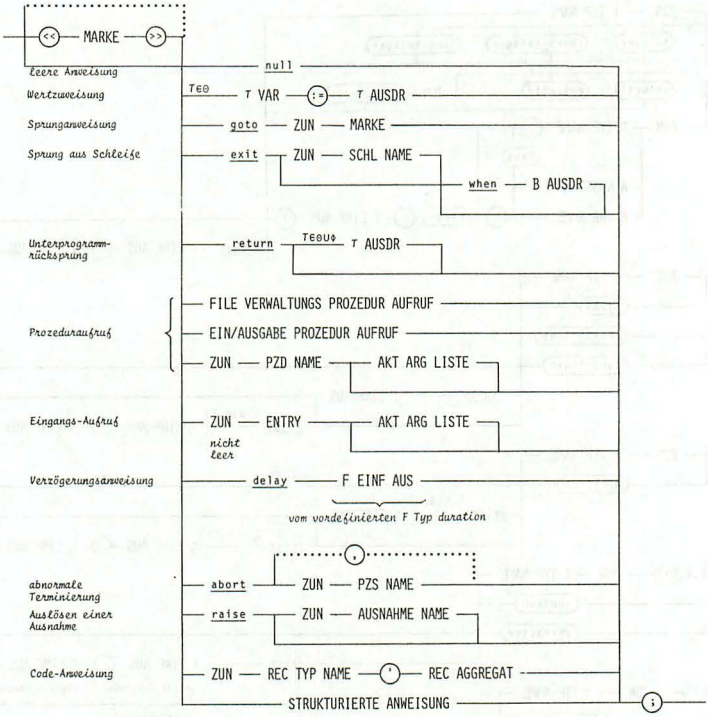
ADA 22: τ TYP $\qquad \tau \in \Theta \cup \{PZS\} \cup \Phi$

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$, $\Phi = \{IF,OF,IOF,ITF,OTF\}$ $\qquad \Theta_D = \{B,CH,AZ,I\}$

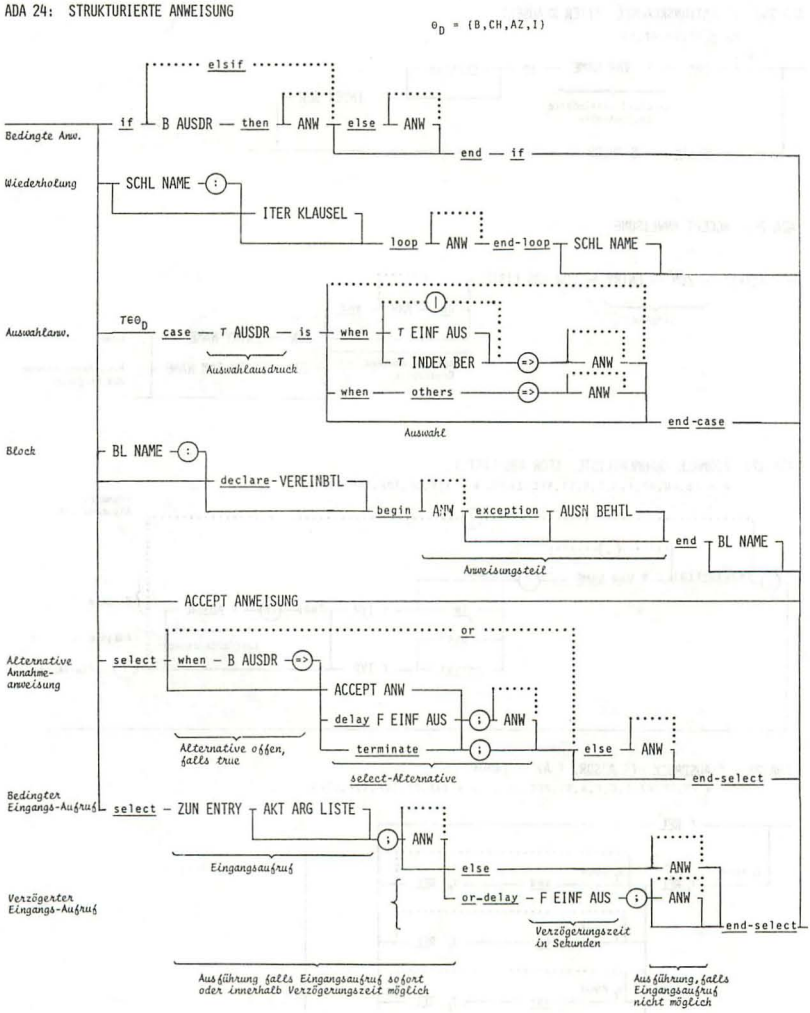


ADA 23: ANWEISUNG (ANW)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , ϕ = {IF,OF,IOF,ITF,OTF}

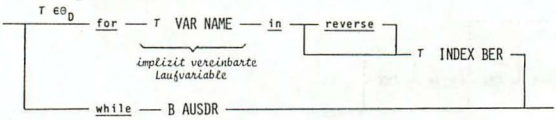


ADA 24: STRUKTURIERTE ANWEISUNG

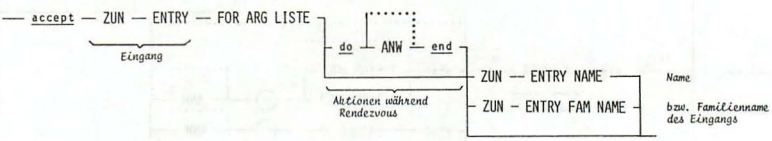


ADA 25: ITERATIONSKLAUSEL (ITER KLAUSEL)

$\Theta_D = \{B, CH, AZ, I\}$

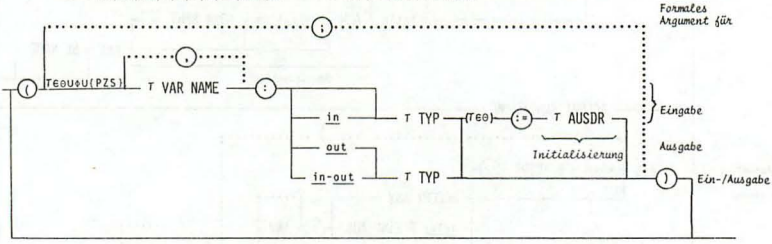


ADA 26: ACCEPT ANWEISUNG



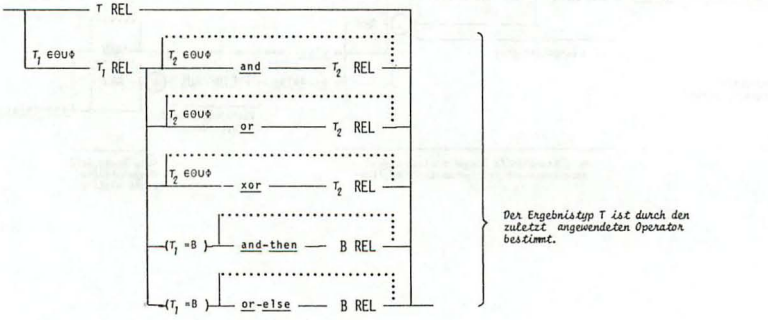
ADA 27: FORMALE ARGUMENTLISTE (FOR ARG LISTE)

$\Theta = \{B, CH, AZ, I, G, F, A, ST, REC, Z\}$ $\Phi = \{IF, OF, IOF, ITF, OTF\}$



ADA 28: T AUSDRUCK (T AUSDR, T A) $T \in \Theta \cup \Phi$

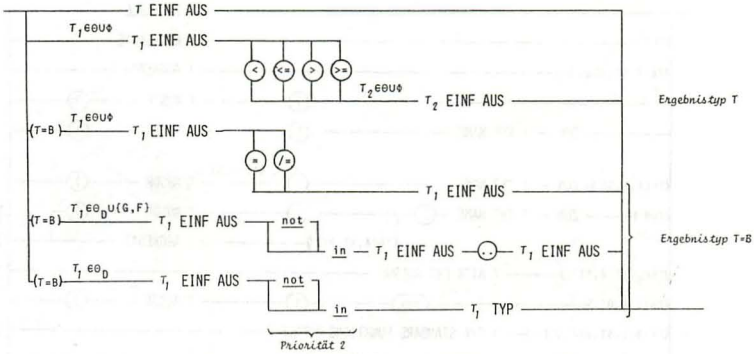
$\Theta = \{B, CH, AZ, I, G, F, A, ST, REC, Z\}$, $\Phi = \{IF, OF, IOF, ITF, OTF\}$



Priorität 1

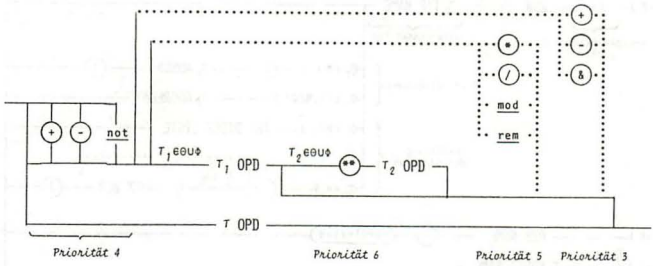
ADA 29: T RELATION (T REL) $T \in \Theta \cup \Phi$

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$, $\Phi = \{IF,OF,IOF,ITF,OTF\}$ $\Theta_D = \{B,CH,AZ,I\}$



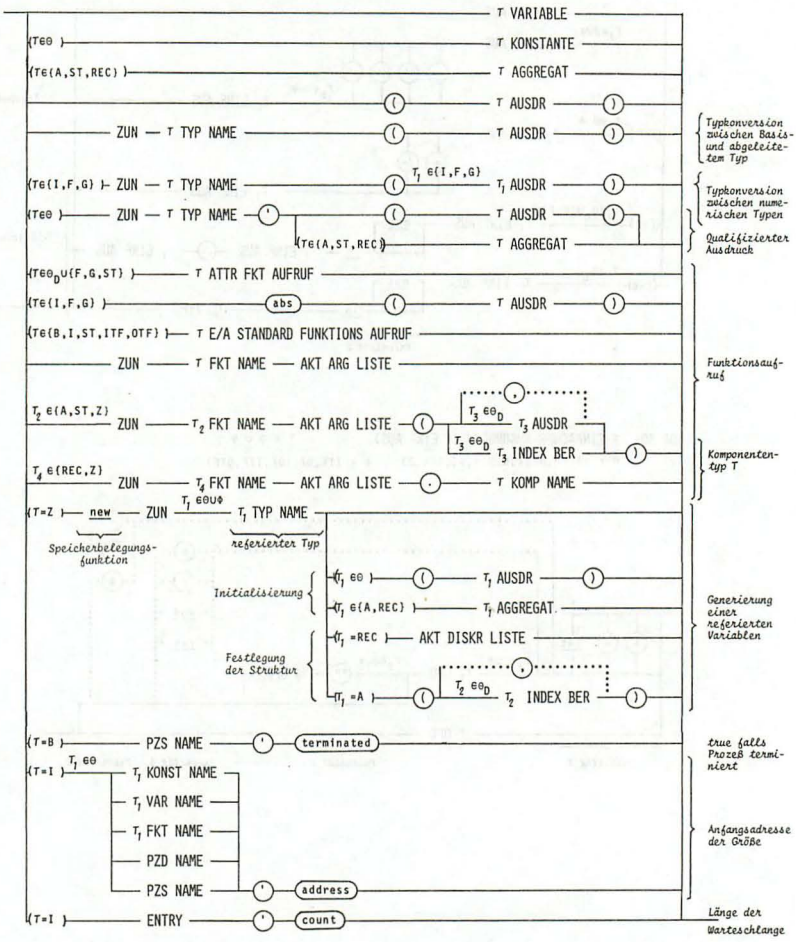
ADA 30: T EINFACHER AUSDRUCK (T EINF AUS) $T \in \Theta \cup \Phi$

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$ $\Phi = \{IF,OF,IOF,ITF,OTF\}$



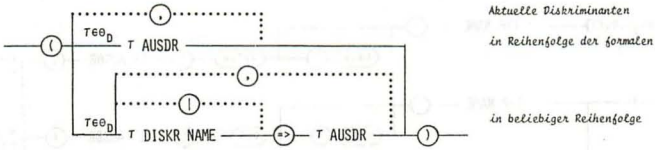
ADA 31: T OPERAND (T OPD) $T \in \Theta \cup \Phi$

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$ $\Phi = \{IF,OF,IOF,ITF,OTF\}$ $\Theta_D = \{B,CH,AZ,I\}$



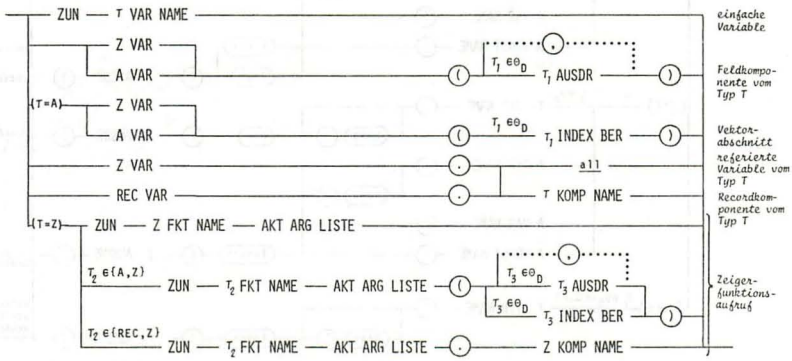
ADA 32: AKTUELLE DISKRIMINANTENLISTE (AKT DISKR LISTE)

Θ_D = {B,CH,AZ,I}



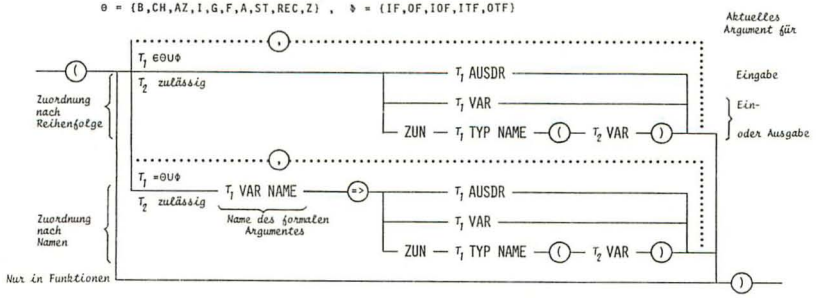
ADA 33: T VARIABLE (T VAR) $T \in \Theta \cup \{PZS\} \cup \Phi$

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF} Θ_D = {B,CH,AZ,I}



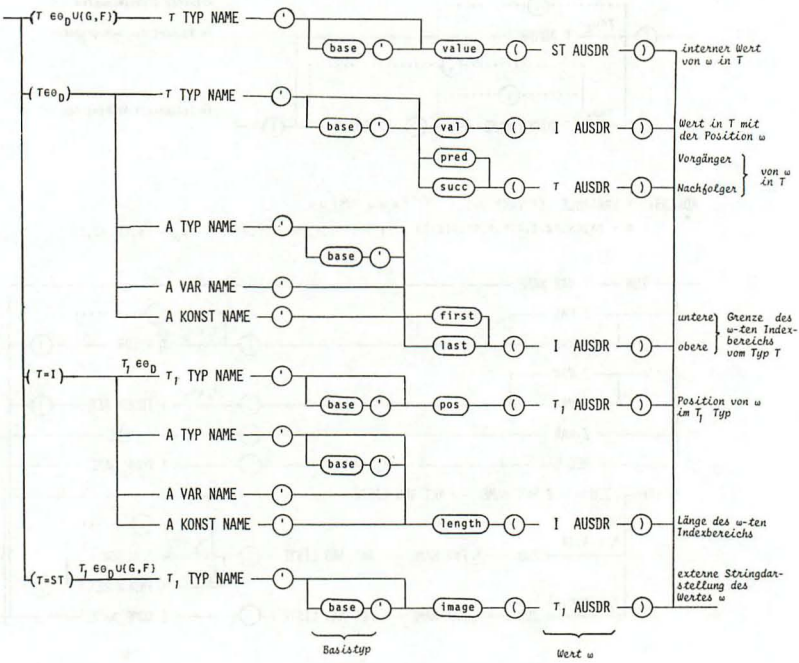
ADA 34: AKTUELLE ARGUMENTLISTE (AKT ARG LISTE)

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} , Φ = {IF,OF,IOF,ITF,OTF}

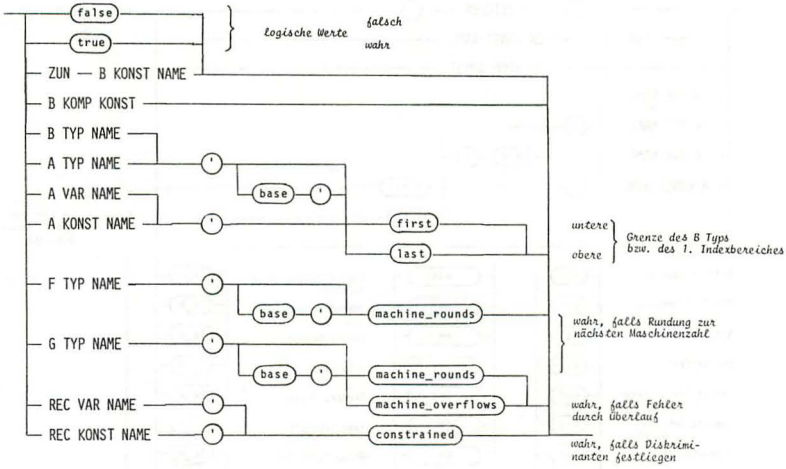


ADA 35: T ATTRIBUTIERTER FUNKTIONSAUFRUF (T ATTR FKT AUFRUF) $T \in \Theta_D \cup \{G,F,ST\}$

$\Theta_D = \{B,CH,AZ,I\}$

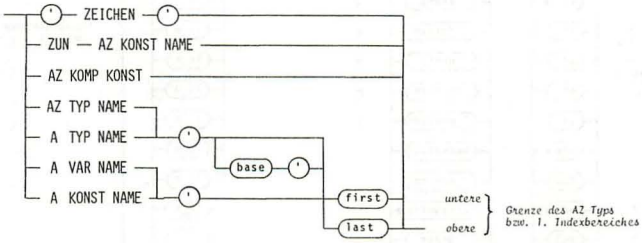


ADA 36: B KONSTANTE (B KONST,BOOLEAN KONSTANTE)

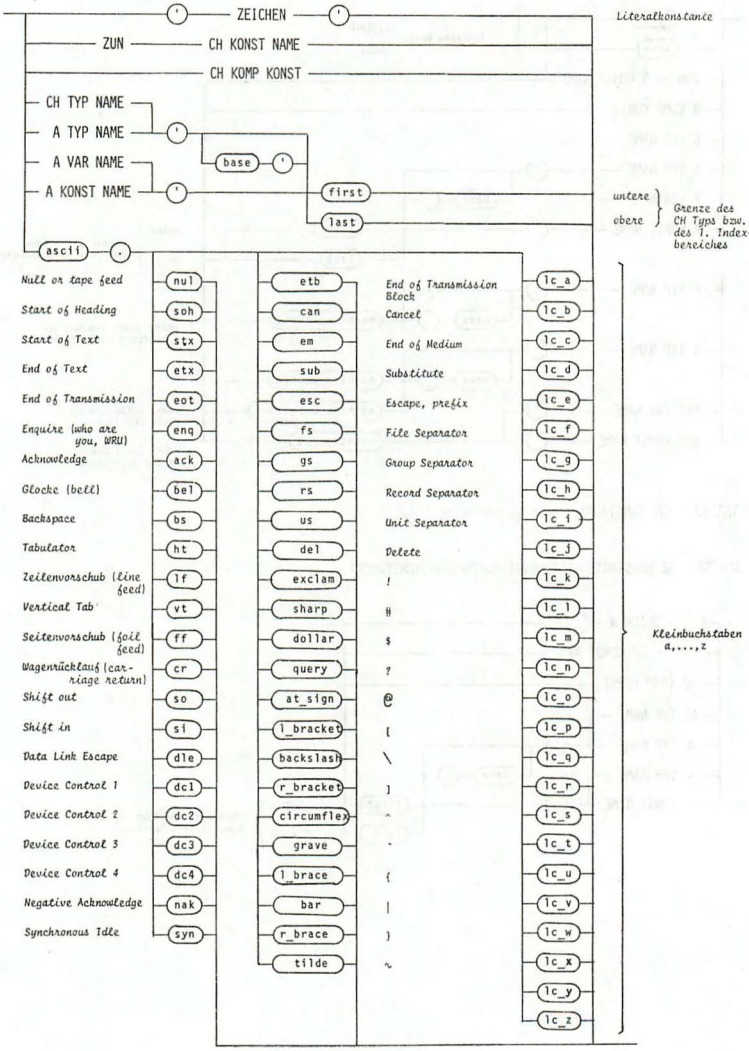


ADA 37: CH KONSTANTE *siehe nächste Seite*

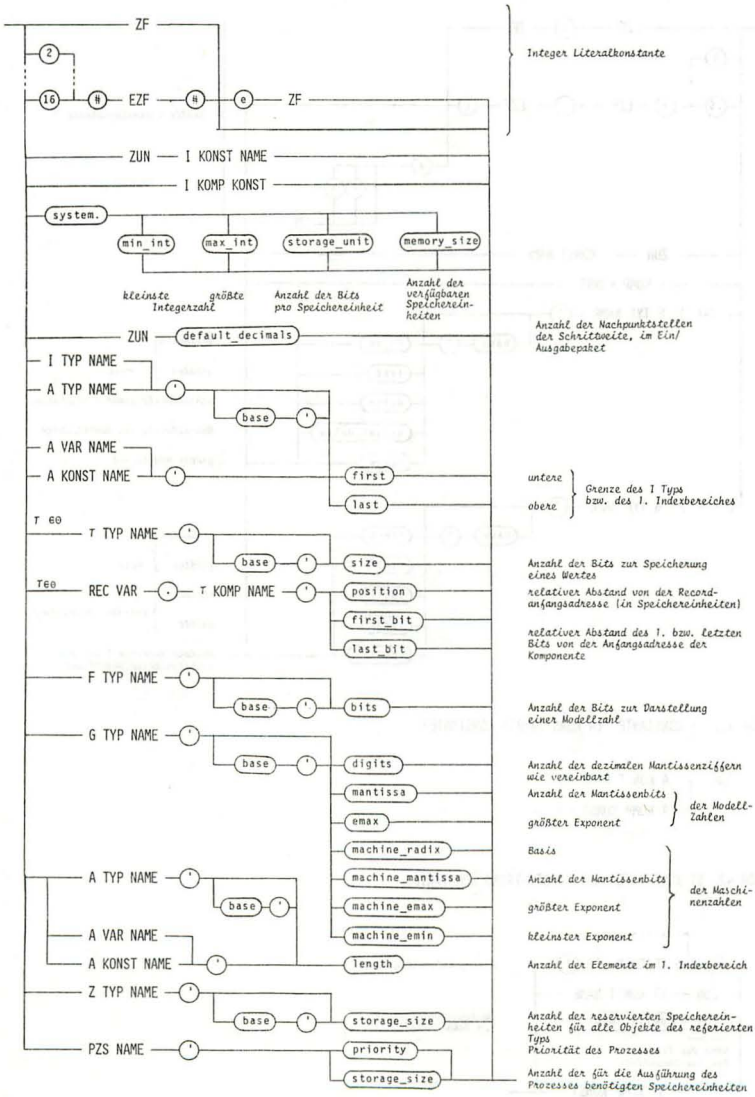
ADA 38: AZ KONSTANTE (AZ KONST,AUFZÄHLUNGSKONSTANTE)



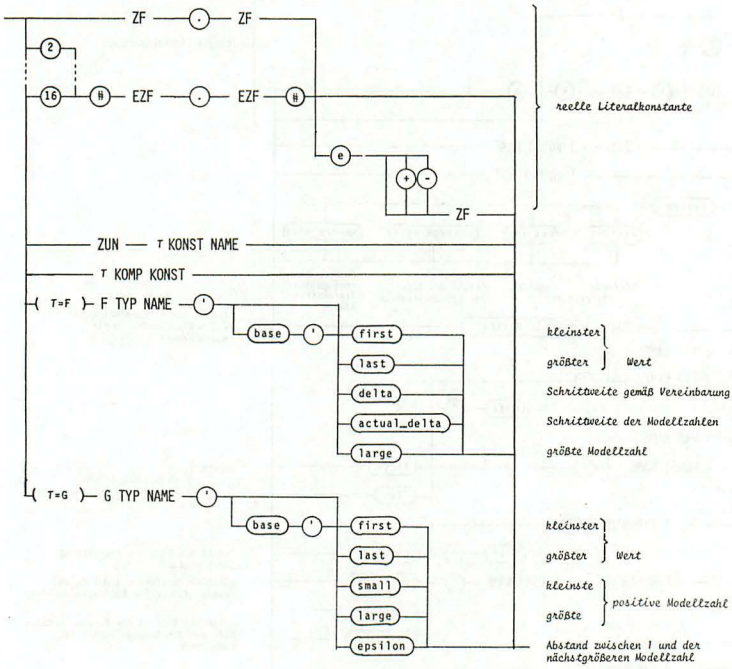
ADA 37: CH KONSTANTE (CH KONST,CHARACTER KONSTANTE)



ADA 39: I KONSTANTE (I KONST,INTEGER KONSTANTE)

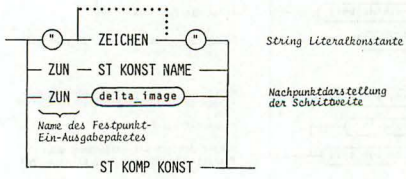


ADA 40: T KONSTANTE (T KONST,GLEITPUNKT KONSTANTE,FESTPUNKT KONSTANTE) T ∈ {G,F}

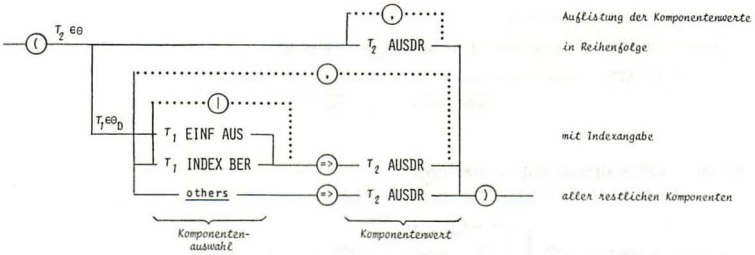


ADA 41: A KONSTANTE (A KONST,ARRAY KONSTANTE)

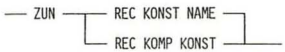
ADA 42: ST KONSTANTE (ST KONST,STRING KONSTANTE)



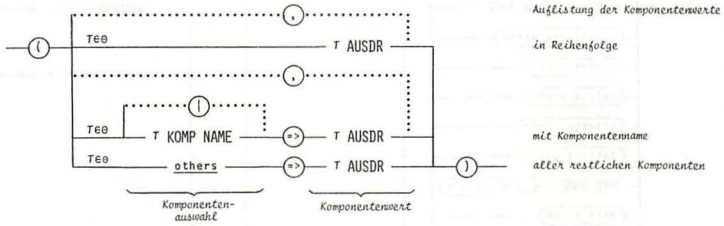
ADA 43: T AGGREGAT T ∈ {A,ST}
θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} θ_D = {B,CH,AZ,I}



ADA 44: REC KONSTANTE (REC KONST,RECORD KONSTANTE)



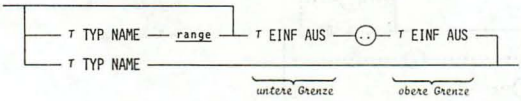
ADA 45: REC AGGREGAT
θ = {B,CH,AZ,I,G,F,A,ST,REC,Z}



ADA 46: Z KONSTANTE (Z KONST,ZEIGER KONSTANTE)

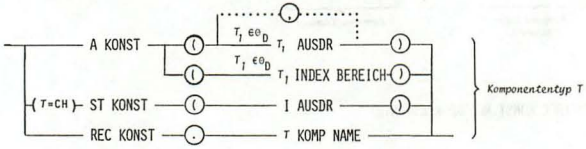
ADA 47: T INDEX BEREICH (T INDEX BER) $T \in \Theta_D$

Θ_D = {B,CH,AZ,I}

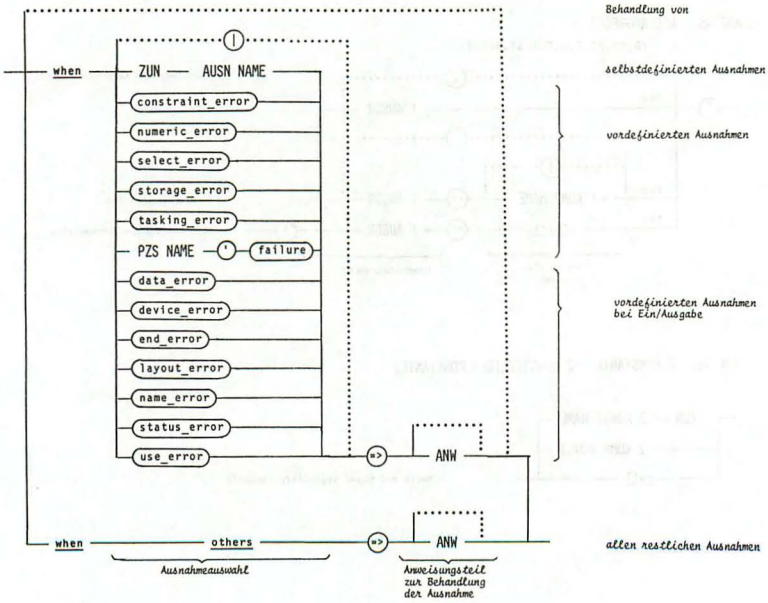


ADA 48: T KOMPONENTENKONSTANTE (T KOMP KONST) $T \in \Theta$

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Θ_D = {B,CH,AZ,I}

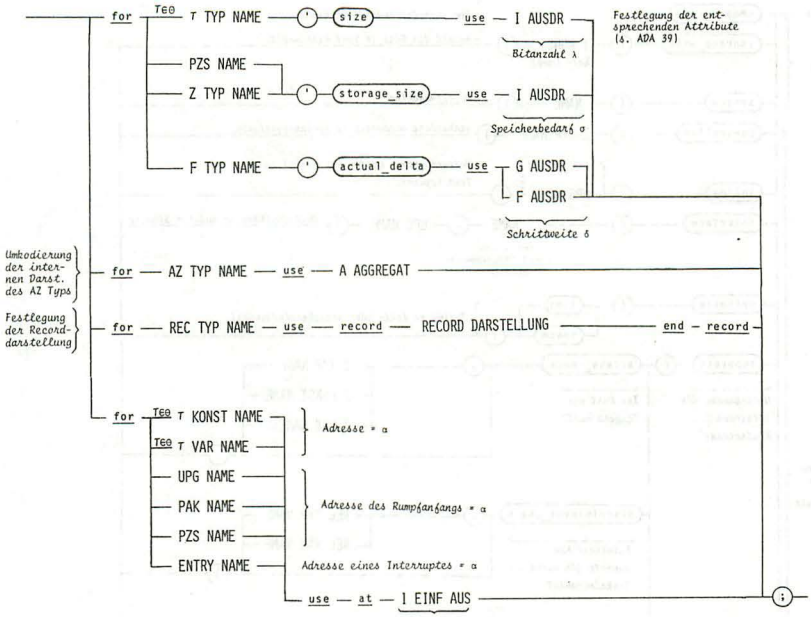


ADA 49: AUSNAHMEBEHANDLUNGSTEIL (AUSN BEHTL)



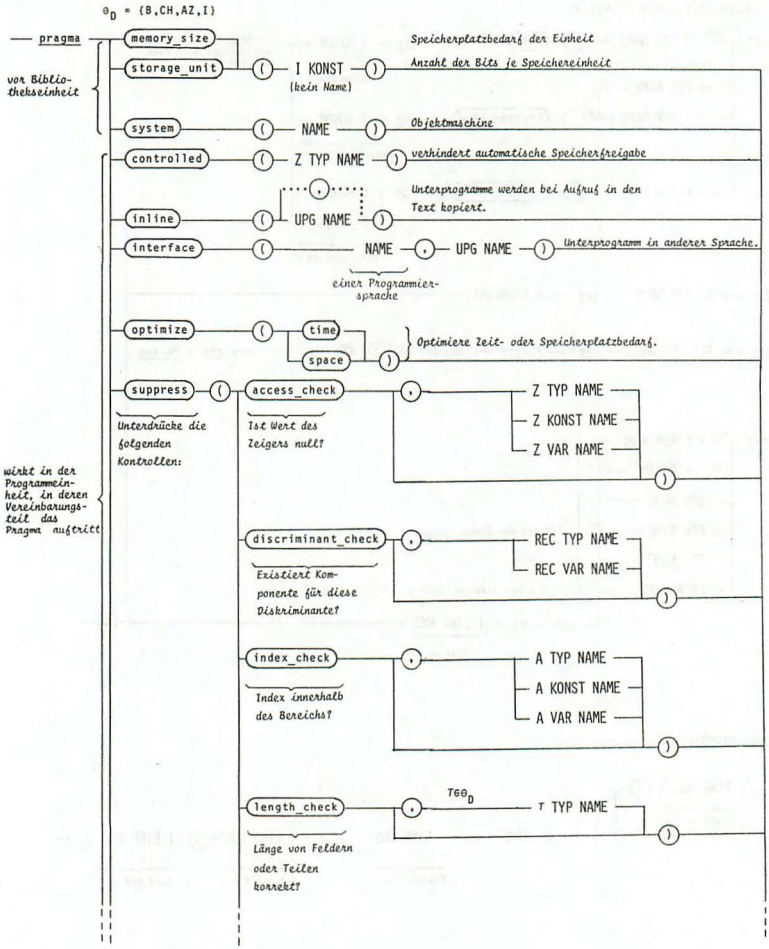
ADA 50: DARSTELLUNGSANGABE (DARST ANG)

θ = {B,CH,AZ,I,G,F,A,ST,REC,Z}

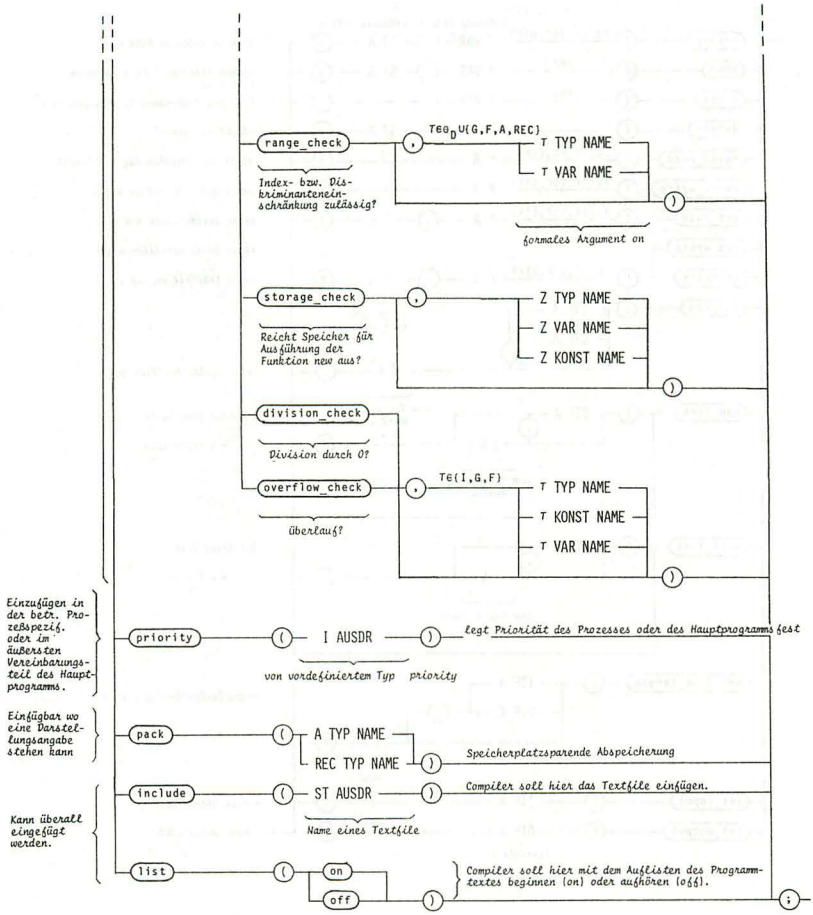


ADA 51: RECORD DARSTELLUNG

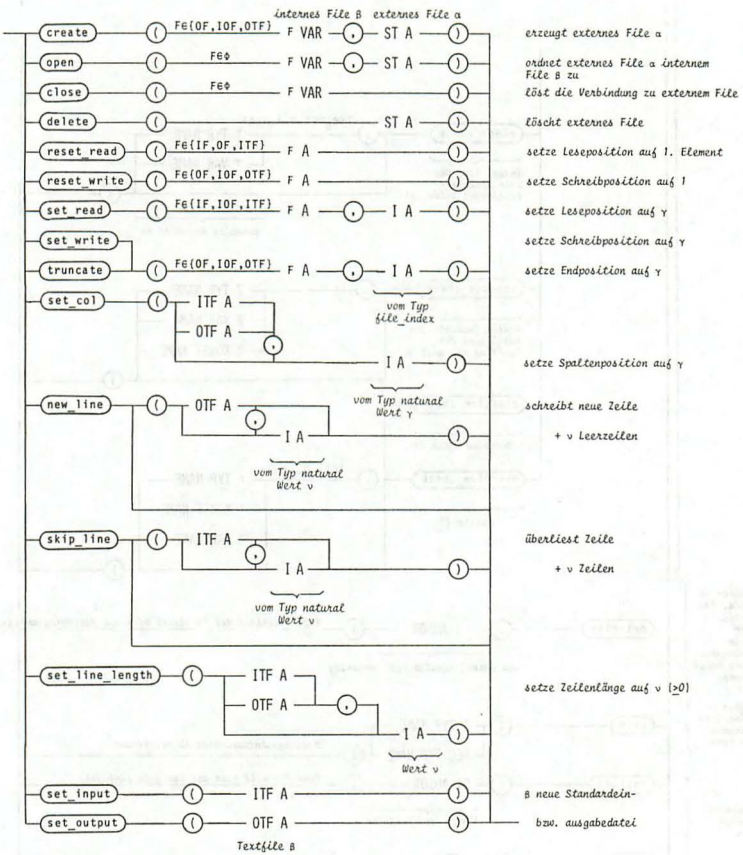
ADA 52: PRAGMA



FORTS. ADA 52

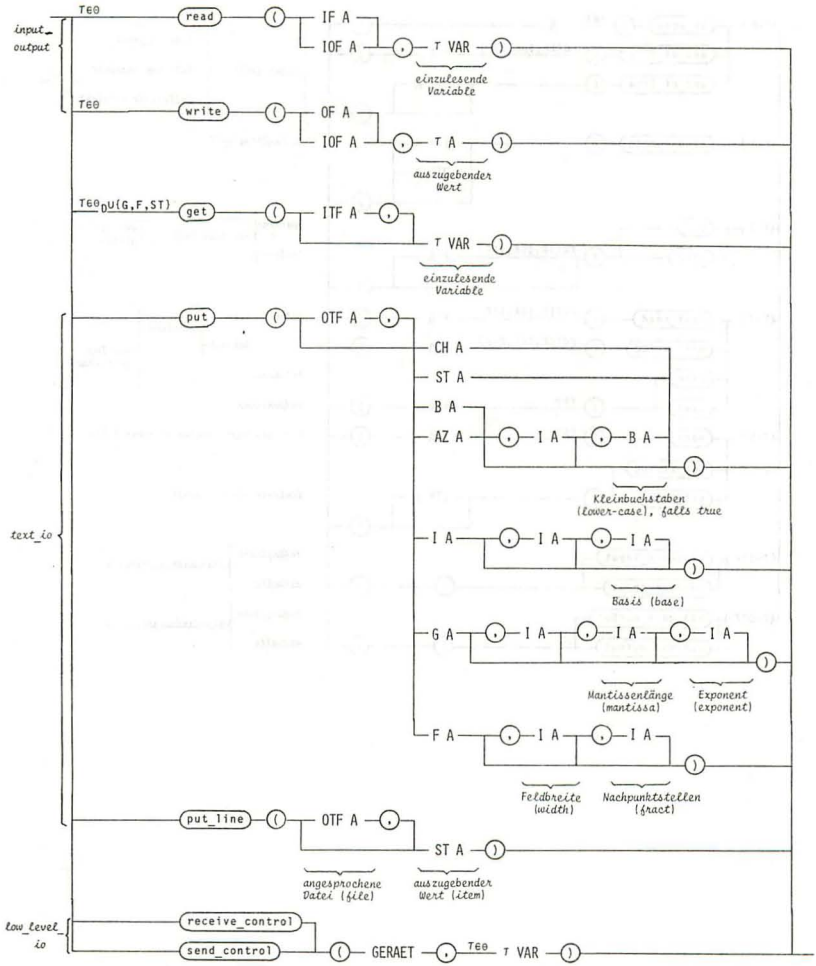

range_check
Index- bzw. Diskriminantenein-schränkung zulässig?
TEQ_D U{G,F,A,REC}
T TYP NAME
T VAR NAME
formales Argument on
storage_check
Reicht Speicher für Ausführung der Funktion new aus?
Z TYP NAME
Z VAR NAME
Z KONST NAME
division_check
Division durch 0?
overflow_check
Überlauf?
TE(I,G,F)
T TYP NAME
T KONST NAME
T VAR NAME
Einzufügen in der betr. Prozeßspezif. oder im äußersten Vereinbarungsteil des Hauptprogramms.
priority
I AUSDR
von vordefiniertem Typ priority
legt Priorität des Prozesses oder des Hauptprogramms fest
Einfügbar wo eine Darstellungsangabe stehen kann
pack
A TYP NAME
REC TYP NAME
Speicherplatzsparende Abspeicherung
Kann überall eingefügt werden.
include
ST AUSDR
Name eines Textfile
Compiler soll hier das Textfile einfügen.
list
on
off
Compiler soll hier mit dem Auflisten des Programmtextes beginnen (on) oder aufhören (off).

ADA 53: FILEVERWALTUNGSPROZEDURAUFRUF $\phi \in \{IF, OF, IOF, ITF, OTF\}$



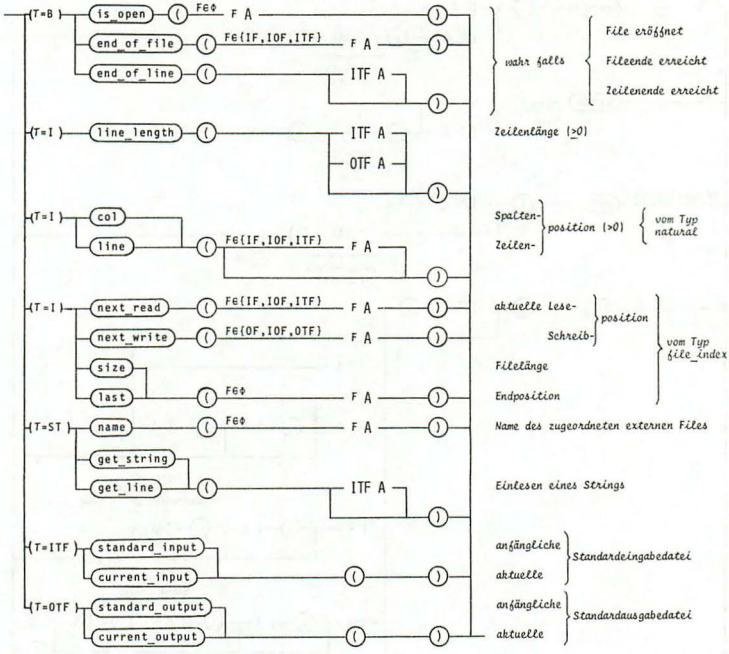
ADA 54: E/A PROZEDURAUFRUF

Θ = {B,CH,AZ,I,G,F,A,ST,REC,Z} Θ_D = {B,CH,AZ,I}



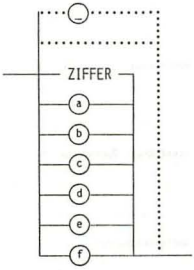
ADA 55: T E/A STANDARDFUNKTIONSAUFRUF Φ = {IF,OF,IOF,ITF,OTF}

T ∈ {B,I,ST,ITF,OTF}

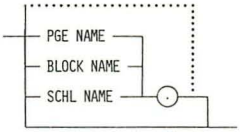


ADA 56: ZIFFERNFOLGE (ZF)

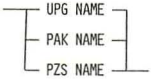
ADA 57: ERWEITERTE ZIFFERNFOLGE (EZF)



ADA 58: ZUNAME (ZUN)



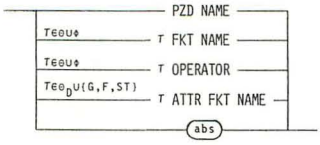
ADA 59: PROGRAMMEINHEIT NAME (PGE NAME)



ADA 60: UNTERPROGRAMM NAME (UPG NAME)

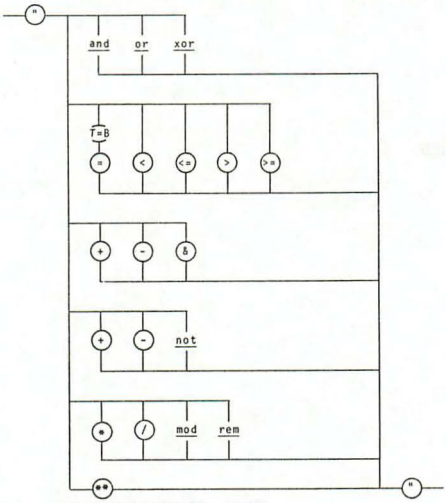
$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$, $\Phi = \{IF,OF,IOF,ITF,OTF\}$

$\Theta_D = \{B,CH,AZ,I\}$



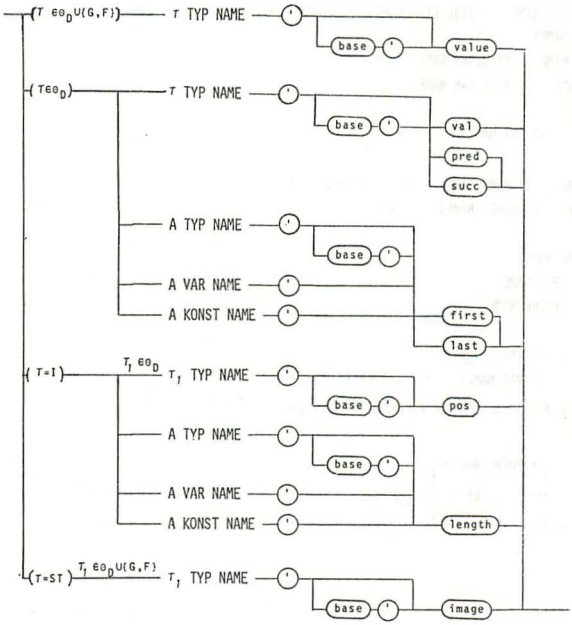
ADA 61: T OPERATOR T ∈ O ∪ Φ

O = {B,CH,AZ,I,G,F,A,ST,REC,Z} Φ = {IF,OF,IOF,ITF,OTF}

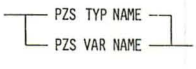


ADA 62: τ ATTRIBUTIERTER FUNKTIONSNAME (τ ATTR FKT NAME) $\tau \in \Theta_D \cup \{G,F,ST\}$

$\Theta_D = \{B,CH,AZ,I\}$

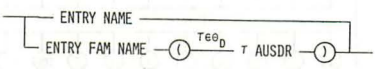


ADA 63: PZS NAME



ADA 64: ENTRY

$\Theta_D = \{B,CH,AZ,I\}$

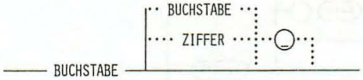


ADA 65: NAME

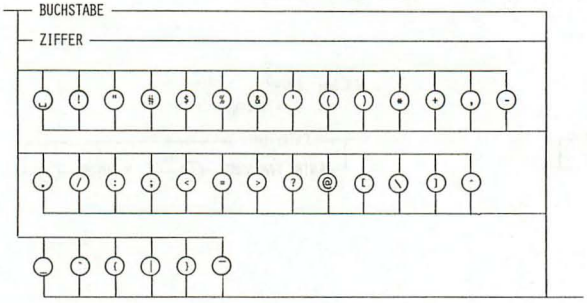
AUSNAHME NAME
BIBLIOTHEKS EINHEITS NAME (BIBL EINH NAME)
BLOCKNAME (BL NAME)
τ DISKRIMINANTEN NAME (τ DISKR NAME) $\tau \in \Theta_D$
ENTRY FAMILIEN NAME (ENTRY FAM NAME)
ENTRY NAME
τ FUNKTIONS NAME (τ FKT NAME) $\tau \in \Theta \cup \Phi$
GERAET
τ KOMPONENTEN NAME (τ KOMP NAME) $\tau \in \Theta \cup \{PZS\} \cup \Phi$
τ KONSTANTEN NAME (τ KONST NAME) $\tau \in \Theta$
MARKE
PAKET NAME (PAK NAME)
PROZEDUR NAME (PZD NAME)
SCHLEIFEN NAME (SCHL NAME)
TYP NAME
τ TYP NAME $\tau \in \Theta \cup \{PZS\} \cup \Phi$
τ VARIABLEN NAME (τ VAR NAME) $\tau \in \Theta \cup \{PZS\} \cup \Phi$

$\Theta = \{B,CH,AZ,I,G,F,A,ST,REC,Z\}$, $\Phi = \{IF,OF,IOF,ITF,OTF\}$

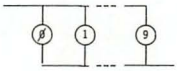
$\Theta_D = \{B,CH,AZ,I\}$



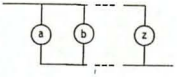
ADA 66: ZEICHEN



ADA 67: ZIFFER



ADA 68: BUCHSTABE



D. VERZEICHNISSE

1. VERZEICHNIS DER SYNTAXDIAGRAMME

ADA	NAME BZW. ABKÜRZUNG	ref. von ADA
28	T A (T AUSDRUCK)	
26	ACCEPT ANWEISUNG	24
45	REC AGGREGAT	23,48
43	T AGGREGAT	31,50
34	AKT ARG LISTE (AKTUELLE ARGUMENTLISTE)	
32	AKT DISKR LISTE (AKTUELLE DISKRIMINANTEN-LISTE)	
12	AKT SCH ARG LISTE (AKTUELLE SCHEMA ARGUMENT LISTE)	
13	AKT SCHEMA ARG (AKTUELLES SCHEMA ARGUMENT)	
34	AKTUELLE ARGUMENTLISTE (AKT ARG LISTE)	23,24,31,33
32	AKTUELLE DISKRIMINANTENLISTE (AKT DISKR LISTE)	31
12	AKTUELLE SCHEMA ARGUMENT LISTE (AKT SCH ARG LISTE)	8,11
13	AKTUELLES SCHEMA ARGUMENT (AKT SCHEMA ARG)	12
23	ANW (ANWEISUNG)	
23	ANWEISUNG (ANW)	6,9,14,24,26,49
35	T ATTR FKT AUFRUF (T ATTRIBUTIERTER FUNKTIONSAUFRUF)	
62	T ATTR FKT NAME (T ATTRIBUTIERTER FUNKTIONSNAME)	
35	T ATTRIBUTIERTER FUNKTIONSAUFRUF (T ATTR FKT AUFRUF)	31
62	T ATTRIBUTIERTER FUNKTIONSNAME (T ATTR FKT NAME)	13,18,60
28	T AUSDR (T AUSDRUCK)	
28	T AUSDRUCK (T AUSDR, T A)	5,13,18,19,21,23,24,25,27,31,32,33,34,35,43,45,48,50,52,53,54,55,64
49	AUSN BEHTL (AUSNAHMEBEHANDLUNGSTEIL)	
49	AUSNAHMEBEHANDLUNGSTEIL (AUSN BEHTL)	6,9,14,24
68	BUCHSTABE	65,66
50	DARST ANG (DARSTELLUNGSANGABE)	
50	DARSTELLUNGSANGABE (DARST ANG)	10,15,17

21	DATENSATZLISTE (DSL)	20,21
21	DSL (DATENSATZLISTE)	20,21
54	E/A PROZEDURAUFRUF	23
55	*T* E/A STANDARDFUNKTIONSAUFRUF	31
30	*T* EINF AUS (*T* EINFACHER AUSDRUCK)	
30	*T* EINFACHER AUSDRUCK (*T* EINF AUS)	20,21,22,23, 24,29,43,47, 50,51
64	ENTRY	18,23,24,26, 31
16	ENTRY VEREINB (ENTRY VEREINBARUNG)	
16	ENTRY VEREINBARUNG (ENTRY VEREINB)	15
57	ERWEITERTE ZIFFERNFOLGE (EZF)	39,40
57	EZF (ERWEITERTE ZIFFERNFOLGE)	39,40
53	FILEVERWALTUNGSPROZEDURAUFRUF	23
27	FOR ARG LISTE (FORMALE ARGUMENTLISTE)	7,16,26
19	FOR DISKR L (FORMALE DISKRIMINANTEN-LISTE)	5,18
5	FOR SCH ARG LISTE (FORMALE SCHEMA ARGU-MENTLISTE)	3,4
27	FORMALE ARGUMENTLISTE (FOR ARG LISTE)	7,16,26
19	FORMALE DISKRIMINANTENLISTE (FOR DISKR L	5,18
5	FORMALE SCHEMA ARGUMENTLISTE (FOR SCH ARG LISTE)	3,4
65	GERAET	54
47	*T* INDEX BER (*T* INDEX BEREICH)	
47	*T* INDEX BEREICH (*T* INDEX BER)	16,20,21,22, 24,25,31,33, 43
25	ITER KLAUSEL (ITERATIONSKLAUSEL)	24
25	ITERATIONSKLAUSEL (ITER KLAUSEL)	24
48	*T* KOMP KONST (*T* KOMPONENTENKONSTANTE)	
48	*T* KOMPONENTENKONSTANTE (*T* KOMP KONST)	37,38,39,40, 41,42,44,46
41	A KONST (A KONSTANTE)	18,31,48
38	AZ KONST (AZ KONSTANTE)	18,31
36	B KONST (B KONSTANTE)	18,31
37	CH KONST (CH KONSTANTE)	18,31
39	I KONST (I KONSTANTE)	18,31
44	REC KONST (REC KONSTANTE)	18,31
42	ST KONST (ST KONSTANTE)	18,31
40	*T* KONST (*T* KONSTANTE)	18,31
46	Z KONST (Z KONSTANTE)	18,31
41	A KONSTANTE (A KONST)	18,31,48
38	AZ KONSTANTE (AZ KONST)	18,31
36	B KONSTANTE (B KONST)	18,31
37	CH KONSTANTE (CH KONST)	18,31
39	I KONSTANTE (I KONST)	18,31,52
44	REC KONSTANTE (REC KONST)	18,31
42	ST KONSTANTE (ST KONST)	18,31,48
40	*T* KONSTANTE (*T* KONST)	18,31

46	Z KONSTANTE (Z KONST)	18,31
2	KONTEXTANGABE	1
65	MARKE, NAME	
65	NAME, MARKE, GERAET	2,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,31,32,33,34,35,36,37,38,39,40,41,42,44,45,46,47,48,49,50,51,52,54,58,59,60,62,63,64
31	*T* OPD (*T* OPERAND)	30
31	*T* OPERAND (*T* OPD)	30
61	*T* OPERATOR	6,7,8,13,18,60
10	PAK SPEZIF (PAKETSPEZIFIKATION)	1,4,17,18
4	PAKET SCHEMA SPEZIFIKATION)	1,18
11	PAKETERZEUGUNG	1,17,18
9	PAKETRUMPF	1,17
10	PAKETSPEZIFIKATION (PAK SPEZIF)	1,4,17,18
59	PGE NAME (PROGRAMMEINHEITSNAME)	1,58
52	PRAGMA	
1	PROGRAMM	
59	PROGRAMMEINHEIT NAME (PGE NAME)	1,58
14	PROZESSRUMPF	1,17
15	PROZESSSPEZIFIKATION	17,18
63	PZS NAME	17,18,23,31,49,50,59
65	PZS TYP NAME	14,15,63
65	PZS VAR NAME	14,15,63
51	RECORD DARSTELLUNG	50
29	*T* REL (*T* RELATION)	
29	*T* RELATION (*T* REL)	28
24	STRUKTURIERTE ANWEISUNG	23
22	*T* TYP	5,7,8,11,13,18,19,20,21,27,29
20	*T* TYP DEF (*T* TYP DEFINITION)	
20	*T* TYP DEFINITION (*T* TYP DEF)	18,21
65	*T* TYP NAME	
6	UNTERPROGRAMM (UPG)	1,17
8	UNTERPROGRAMM ERZEUGUNG	1,18
60	UNTERPROGRAMM NAME (UPG NAME)	52,59
3	UNTERPROGRAMM SCHEMA SPEZIFIKATION	1,18
7	UNTERPROGRAMM SPEZIFIKATION (UPG SPEZIF)	1,3,5,6,17,18
6	UPG (UNTERPROGRAMM)	

60	UPG NAME (UNTERPROGRAMM NAME)	
7	UPG SPEZIF (UNTERPROGRAMM SPEZIFIKATION)	
33	*T* VAR (*T* VARIABLE)	
33	*T* VARIABLE (*T* VAR)	13,18,23,31, 33,34,39,53, 54
18	VEREINB (VEREINBARUNG)	
18	VEREINBARUNG (VEREINB)	10,17
17	VEREINBARUNGSTEIL (VEREINBTL)	6,9,14,24
17	VEREINBTL (VEREINBARUNGSTEIL)	
66	ZEICHEN	20,37,38,42
56	ZF (ZIFFERNFOLGE)	
67	ZIFFER	56,57,65,66
56	ZIFFERNFOLGE (ZF)	39,40
58	ZUN (ZUNAME)	
58	ZUNAME (ZUN)	1,8,11,13,18, 20,22,23,24, 26,31,33,34, 36,37,38,39, 40,41,42,44, 46,49

2. VERZEICHNIS DER WORTSYMBOLE

abort ADA 23
accept ADA 26
access ADA 20
all ADA 33
and ADA 28,61
array ADA 20
at ADA 50,51
begin ADA 6,9,14,24
body ADA 9,14,17
case ADA 21,24
constant ADA 18
declare ADA 24
delay ADA 23,24
delta ADA 5,20,22
digits ADA 5,20,22
do ADA 26
else ADA 24,28
elsif ADA 24
end ADA 6,9,10,14,15,21,24,26,50
entry ADA 16
exception ADA 6,9,14,18,24
exit ADA 23
for ADA 25,50
function ADA 7,8
generic ADA 3,4
goto ADA 23
if ADA 24
in ADA 5,6,8,25,27,29
is ADA 5,6,8,9,10,11,14,15,17,18,21,24
limited ADA 5,20
loop ADA 24
mod ADA 30,51,61

new ADA 8,11,20,31
not ADA 29,30,61
null ADA 21,23,46
of ADA 20
or ADA 24,28,61
others ADA 21,24,43,45,49
out ADA 5,27
package ADA 9,10,11,17,18
pragma ADA 52
private ADA 5,10,20
procedure ADA 7,8
raise ADA 23
range ADA 5,20,22,47,51
record ADA 20,50
rem ADA 30,61
renames ADA 18
return ADA 7,23
reverse ADA 25
select ADA 24
separate ADA 1,17
subtype ADA 18
task ADA 14,15,17,18
terminate ADA 24
then ADA 24,28
type ADA 5,15,18
use ADA 2,18,50
when ADA 21,23,24,49
while ADA 25
xor ADA 28,61

3. VERZEICHNIS DER STANDARDNAMEN

(in Ovale eingeschlossene Buchstabenfolgen)

Bei der nachfolgenden Angabe der Gruppen wurden folgende Abkürzungen verwendet:

A	Attribut	Pak	Paket
Aus	Ausnahme	SF	Funktion
K	Konstante	SP	Prozedur
P	Pragma	T	Typ

Name	Gruppe	Diagramm
abs	SF	ADA 31,60
ack	K	ADA 37
actual_delta	A	ADA 40,50
address	A	ADA 31
ascii	Pak	ADA 37
at_sign	K	ADA 37
backslash	K	ADA 37
bar	K	ADA 37
base	A	ADA 22,35,36,37,38,39,40,62
bel	K	ADA 37
bits	A	ADA 39
boolean	T	ADA 22
bs	K	ADA 37
can	K	ADA 37
character	T	ADA 22
circumflex	K	ADA 37
close	SP	ADA 53
col	SF	ADA 55
constrained	A	ADA 36
constraint_error	Aus	ADA 49
controlled	P	ADA 52
count	A	ADA 31
cr	K	ADA 37
create	SP	ADA 55
current_input	SF	ADA 55
current_output	SF	ADA 55
data_error	Aus	ADA 49
dc1	K	ADA 37
dc2	K	ADA 37
dc3	K	ADA 37
dc4	K	ADA 37

default_decimals	K	ADA 39
del	K	ADA 37
delete	SP	ADA 53
delta	A	ADA 40
delta_image	K	ADA 42
device_error	Aus	ADA 49
digits	A	ADA 39
dle	K	ADA 37
dollar	K	ADA 37
duration	T	ADA 22
em	K	ADA 37
emax	A	ADA 39
end_error	Aus	ADA 49
end_of_file	SF	ADA 55
end_of_line	SF	ADA 55
enq	K	ADA 37
eot	K	ADA 37
epsilon	A	ADA 40
esc	K	ADA 37
etb	K	ADA 37
etx	K	ADA 37
exclam	K	ADA 37
failure	A	ADA 49
false	K	ADA 36,49
ff	K	ADA 37
file_index	T	ADA 22
first	A	ADA 35,36,37,38,39,40,62
first_bit	A	ADA 39
fixed_io	Pak	ADA 11
float	T	ADA 22
float_io	Pak	ADA 11
fs	K	ADA 37
get	SP	ADA 54
get_line	SF	ADA 55
get_string	SF	ADA 55
gs	K	ADA 37
grave	K	ADA 37
ht	K	ADA 37
image	A	ADA 35,62
include	P	ADA 52
inline	P	ADA 52
inout_file	T	ADA 22
input_output	Pak	ADA 11
integer	T	ADA 22
integer_io	Pak	ADA 11
interface	P	ADA 52
in_file	T	ADA 22
is_open	SF	ADA 55
large	A	ADA 40
last	A	ADA 35,36,37,38,39,40,62
	SF	ADA 55

last_bit	A	ADA 39
layout_error	Aus	ADA 49
lc_a	K	ADA 37
⋮		
lc_z	K	ADA 37
length	A	ADA 35,39,62
lf	K	ADA 37
line	SF	ADA 55
line_length	SF	ADA 55
list	P	ADA 52
long_float	T	ADA 22
long_integer	T	ADA 22
l_brace	K	ADA 37
l_bracket	K	ADA 37
machine_emax	A	ADA 39
machine_emin	A	ADA 39
machine_mantissa	A	ADA 39
machine_overflows	A	ADA 39
machine_radix	A	ADA 39
machine_rounds	A	ADA 36
mantissa	A	ADA 39
max_int	K	ADA 39
memory_size	K	ADA 39
	P	ADA 52
min_int	K	ADA 39
nak	K	ADA 37
name	SF	ADA 55
name_error	Aus	ADA 49
natural	T	ADA 22
new_line	SP	ADA 53
next_read	SP	ADA 55
next_write	SP	ADA 55
nul	K	ADA 37
numeric_error	Aus	ADA 49
open	SP	ADA 53
optimize	P	ADA 52
out_file	T	ADA 22
pack	P	ADA 52
pos	A	ADA 35,62
position	A	ADA 39
pred	A	ADA 35,62
priority	T	ADA 39,22
	P	ADA 52
put	SP	ADA 54
put_line	SP	ADA 54
query	K	ADA 37
range	A	ADA 22
read	SP	ADA 54
receive_control	SP	ADA 54
reset_read	SP	ADA 53

reset_write	SP	ADA 53
rs	K	ADA 37
r_brace	K	ADA 37
r_bracket	K	ADA 37
select_error	Aus	ADA 49
send_control	SP	ADA 54
set_col	SP	ADA 53
set_input	SP	ADA 53
set_line_length	SP	ADA 53
set_output	SP	ADA 53
set_read	SP	ADA 53
set_write	SP	ADA 53
shared_variable_update	SP	ADA 8
sharp	K	ADA 37
short_float	T	ADA 22
short_integer	T	ADA 22
si	K	ADA 37
size	A	ADA 39,50
	SF	ADA 55
skip_line	SP	ADA 53
small	A	ADA 40
so	K	ADA 37
soh	K	ADA 37
standard_input	SF	ADA 55
standard_output	SF	ADA 55
status_error	Aus	ADA 49
storage_error	Aus	ADA 49
storage_size	A	ADA 39,50
storage_unit	P	ADA 52
	K	ADA 39
string	T	ADA 22
stx	K	ADA 37
sub	K	ADA 37
succ	A	ADA 35,62
suppress	P	ADA 52
syn	K	ADA 37
system	Pak	ADA 39
	P	ADA 52
tasking_error	Aus	ADA 49
terminated	A	ADA 31
text.io	Pak	ADA 11
tilde	K	ADA 37
true	K	ADA 36
truncate	SP	ADA 53
unchecked_conversion	SF	ADA 8
unchecked_deallocation	SP	ADA 8
us	K	ADA 37
use_error	Aus	ADA 49
val	A	ADA 35,62
value	A	ADA 35,62
vt	K	ADA 37
write	SP	ADA 54

4. DAS ADA-WÖRTERBUCH

Syntaxvariable im Reference Manual	Übersetzung (falls vorgenommen)	Diagramm-Nr.
A		
abort statement	abnormale Terminierung	ADA 23
accept statement	ACCEPT ANWEISUNG	ADA 26
access type definition	Z TYPDEFINITION	ADA 20
accuracy constraint	Genauigkeitseinschränkung	ADA 20
actual parameter	aktuelles Argument	ADA 34
actual parameter part	aktuelle Argumentliste	ADA 34
adding operator	Additionsoperator	ADA 30
address specification	Adreßangabe	ADA 50
aggregate	T AGGREGAT	ADA 43,45
alignment clause	———	ADA 51
allocator	Speicherbelegungsfunktion	ADA 31
argument	Argument	ADA 52
array type definition	A TYPDEFINITION	ADA 20
assignment statement	Wertzuweisung	ADA 23
attribute	ATTRIBUTIERTER FUNKTIONS-AUFRUF oder attributierte T KONSTANTE	ADA 31/35,..,40
B		
base	Basis (eines Zahlsystems)	ADA 40
based integer	ERWEITERTE ZIFFERNFOLGE	ADA 57
based number	Basisbezogene Zahl	ADA 40
basic loop	———	ADA 24
block	Block	ADA 24
body	Rumpf	ADA 17
body stub	Rumpfstumpf	ADA 17
C		
case statement	Auswahlanweisung	ADA 24
character literal	Zeichenliteral	ADA 37

character string	Zeichenkette	ADA 42
choice	Auswahl	ADA 21
code statement	Code Anweisung	ADA 23
compilation	PROGRAMM	ADA 1
compilation unit	Übersetzungseinheit	ADA 1
component association	———	ADA 43,45
component declaration	Komponentenvereinbarung	ADA 21
component list	DATENSATZLISTE	ADA 21
compound statement	STRUKTURIERTE ANWEISUNG	ADA 24
condition	Bedingung	ADA 28
conditional entry call	Bedingter Eingangsaufruf	ADA 24
constraint	Einschränkung	ADA 22
context specification	KONTEXTANGABE	ADA 2

D

decimal number	Dezimalzahl	ADA 39,40
declaration	VEREINBARUNG	ADA 18
declarative item	VEREINBARUNG	ADA 18
declarative part	VEREINBARUNGSTEIL	ADA 17
delay statement	Verzögerungsanweisung	ADA 23
derived type definition	Definition eines abgeleiteten Typs	ADA 20
designator	UNTERPROGRAMMNAME	ADA 60
discrete range	INDEXBEREICH	ADA 47
descriminant constraint	AKTUELLE DISKRIMINANTENLISTE	ADA 32
discriminant declaration	———	ADA 32
discriminant part	———	ADA 32
discriminant specification	———	ADA 32

E

entry call	Eingangsaufruf	ADA 23
entry declaration	ENTRY VEREINBARUNG	ADA 16

enumeration literal	AZ KONSTANTE	ADA 38
enumeration type definition	AZ TYPDEFINITION	ADA 20
enumeration type representation	Aufzählungstypdarstellung	ADA 50
exception choice	Ausnahmeauswahl	ADA 46
exception declaration	Ausnahmevereinbarung	ADA 18
exception handler	Ausnahmebehandler	ADA 46
exit statement	Sprung aus Schleife	ADA 23
exponent	Exponent	ADA 40
exponentiating operator	Exponentiationsoperator	ADA 30
expression	AUSDRUCK	ADA 28
extended digit	———	ADA 57

F

factor		ADA 30
fixed point constraint	F TYPDEFINITION	ADA 20
floating point constraint	G TYPDEFINITION	ADA 20
formal parameter	formales Argument	ADA 27
formal part	FORMALE ARGUMENTLISTE	ADA 27
function call	Funktionsaufruf	ADA 31,33

G

generic actual parameter	AKTUELLES SCHEMAARGUMENT	ADA 13
generic association	———	ADA 12
generic formal parameter	Formales Schemaargument	ADA 5
generic instantiation	Ausprägung, Erzeugung	ADA 8,11
generic package declaration	PAKETSCHEMAVEREINBARUNG	ADA 4
generic package instantiation	PAKETERZEUGUNG	ADA 11

generic part		ADA 3,4
generic subprogram declaration	UNTERPROGRAMM SCHEMA VEREINBARUNG	ADA 3
generic subprogram instantiation	UNTERPROGRAMMERZEUGUNG	ADA 8
generic type definition	———	ADA 5
goto statement	Sprunganweisung	ADA 23
I		
identifier	NAME	ADA 65
identifier list	———	ADA 18
if statement	Bedingte Anweisung	ADA 24
imcomplete declaration	Unvollständige Typvereinbarung	ADA 18
index	Index (eines Feldes)	ADA 20
index constraint	Indexeinschränkung	ADA 20,31
indexed component	Feldkomponente	ADA 33
integer	ZIFFERNFOLGE (ZF)	ADA 56
integer type definition	I TYPDEFINITION	ADA 20
iteration clause	ITERATIONSKLAUSEL	ADA 25
L		
label	Marke	ADA 23
length specification	Längenangabe	ADA 50
letter	BUCHSTABE	ADA 68
letter or digit	———	
literal	Literalkonstante	ADA 38,39, 40,42,46
location	Speicherplatz (einer Recordkomponente)	ADA 51
logical operator	logischer Operator	ADA 28
loop parameter	Laufvariable	ADA 25
loop statement	Wiederholungsanweisung	ADA 24
lower case letter	Kleinbuchstabe	ADA 68

M

mode	Argumentübergabemodus	ADA 27
multiplying operator	Multiplikationsoperator	ADA 30

N

name	VARIABLE oder attributierte Konstante	ADA 33, 36,..,40
null statement	Leere Anweisung	ADA 23
number declaration	———	ADA 18
numeric literal	numerische Literalkonkonstante	ADA 39,40

O

object declaration	Variablen-bzw. Konstantenvereinbarung	ADA 18
operator symbol	OPERATOR	ADA 61

P

package body	PAKETRUMPF	ADA 9
package declaration	Paketvereinbarung	ADA
package specification	PAKETSPEZIFIKATION	ADA 10
parameter association	———	ADA 34
parameter declaration	———	ADA 27
pragma	PRAGMA	ADA 52
primary	OPERAND	ADA 31
private type definition	Definition eines geschützten Typs	ADA 20
procedure call	Prozeduraufruf	ADA 23
program component	———	ADA 17
qualified expression	Qualifizierter Ausdruck	ADA 31

R

raise statement	Ausnahmeauslösung	ADA 23
range	Bereich	ADA 20,22, 47
range constraint	Bereichseinschränkung	ADA 20,22, 47

real type definition	R TYPDEFINITION	ADA 20
record type definition	REC TYPDEFINITION	ADA 20
record type representation	Recorddarstellung	ADA 51
relation	RELATION	ADA 29
relational operator	Vergleichsoperator	ADA 29
renaming declaration	Umbennung	ADA 18
representation specification	DARSTELLUNGSANGABE	ADA 50
return statement	Unterprogrammrücksprung	ADA 23

S

select alternative	select-Alternative	ADA 24
selected component	———	
select statement	alternative Annahmeanweisung	ADA 24
selective wait	alternative Annahmeanweisung	ADA 24
sequence of statements	Anweisungsfolge	ADA 6,9, 14,24,26
simple expression	EINFACHER AUSDRUCK	ADA 30
slice	Vektorabschnitt	ADA 33
statement	ANWEISUNG	ADA 23
subprogram body	UNTERPROGRAMM	ADA 6
subprogram declaration	———	ADA 1
subprogram specification	UNTERPROGRAMMSPEZIFICATION	ADA 7
subtype declaration	Untertypvereinbarung	ADA 18
subtype indication	TYP	ADA 22
subunit	Untereinheit	ADA 1
subunit body	———	ADA 1

T

task body	PROZESSRUMPF	ADA 14
task declaration	PROZESSPEZIFIKATION	ADA 15
task specification	PROZESSPEZIFIKATION	ADA 15

term	———	ADA 30
timed entry call	verzögerter Eingangs-aufruf	ADA 24
type conversion	Typkonversion	ADA 31
type declaration	Typvereinbarung	ADA 18
type definition	TYPDEFINITION	ADA 20
type mark	TYP NAME	ADA 22,35

U

unary operator	einstelliger Operator	ADA 30
underscore	Unterstrich	
upper case letter	Großbuchstabe	ADA 68
use clause	use-Klausel	ADA 2

V

variant part	Variantenteil	ADA 21
with clause	with-Klausel	ADA 2

5. STICHWORTVERZEICHNIS

A

Anweisung(-s)
-, Annahme- B.9.2.3, ADA 26
-, abort- B.9.5, ADA 23
-, accept- B.4.11,B.9.2.3, ADA 26
-, Auswahl- B.4.8
-, bedingte B.4.4
-, case- B.4.8
-, code- B.4.13
-, delay- B.9.4, ADA 23
-, Ein- bzw. Ausgabe- B.4.3, B.11
-, exit- B.4.5
-, goto- B.4.6
-, leere B.4.1
-, raise- B.4.12, B.7.3, ADA 23
-, return- B.4.7, B.2.5, ADA 23
-, select- B.9.3
-, Sprung- B.4.6
-, strukturierte B.4
- teil B.4, ADA 29
-, terminate- B.9.5, ADA 23
-, Wiederholungs- B.4.5
-, zur Syndronisierung von Prozeduren B.4.11
abgeleiteter Typ B.2.1.5, ADA 20
Abhängigkeit eines Prozessors B.9
Akürzungen von Typen B.2.1
Abortanweisung B.9.5, ADA 23
accept-Anweisung B.2.6, B.9.2.3, ADA 26
Adreßangabe B.8.2.4, ADA 50
Aggregat B.3.6, B.3.7, ADA 43, ADA 45

Aktivierung eines Prozessors B.9.1
aktuelle(-s)
- Argument(-Liste) B.2.5, B.5.6, ADA 34
- Schemaargument B.2.5, B.6.3, ADA 13
- Schemaargument Liste B.2.5, B.6.3, ADA 12
Alphabet A.2., A.2.1
Alternative B.4.4, ADA 24
- accept-Anweisung B.4.11
-, geschlossene B.9.3
-, offene B.9.3, B.9.4
-, terminate- B.9.3
Anfangswert B.2.2, ADA 19, ADA 21
Annahmeanweisung B.9.2.3, ADA 26
Anonymer Typ B.2.1.1.1.4
Argument B.1.1
-, aktuelles B.5.6, ADA 34
-, aktuelles Schema- B.6.3, ADA 13
-, Ausgabe- B.5.1, ADA 27
-, formales B.5, ADA 22
-, formales Schema- B.6.1, ADA 5
Argumentliste
-, aktuelle B.5.6, ADA 34
-, aktuelle Schema- B.6.3, ADA 12
-, formale B.5, ADA 27
-, formale Schema- B.6.1, ADA 5
Array
- ausdruck B.3.6, ADA 28
-, dynamisch B.2.1.2.1
- typ B.2.1.2.1
- variable B.2.2
Attribute B.2.1.5, B.3.11, ADA 35,36,38,39,40,50
-, Prozeß- B.9.8
Aufruf B.1.1
-, bedingter Eingangs- B.4.11, B.9.3, ADA 24

- einer Funktion B.2.1.5, B.5.4, ADA 31
- einer Prozedur B.2.1.5, B.4.10, B.5.3, ADA 23
- eines Operators B.2.1.5, B.5.5, ADA 28, 29, 30
-, verzögerter Eingangs- B.4.11, B.9.4, ADA 24
Aufzählungs
- typ B.2.1.1.1.1, ADA 20
- darstellung B.8.2.2, ADA 50
- konstante → Konstante ADA 38
Ausdruck B.3, ADA 28
-, Auswahl- B.4.8
- für Fehler B.3.6, ADA 28
- für Files B.11.4, ADA 28
- für Zeichenketten B.3.6.3, ADA 28
- für Zeiger B.3.8, ADA 28
-, ganzzahliger B.3.4, ADA 28
-, logischer B.3.1, ADA 28
-, Record- B.3.7, ADA 28
-, reeller B.3.5, ADA 28
-, statischer B.3.
- vom Aufzählungstyp B.3.3, ADA 28
-, Zeichen- B.3.2, ADA 28
Ausführung eines Programmes B.1, B.1.1, B.10.4
Ausgabe
- anweisung B.4.3, B.11, ADA 54
- argument B.1.1, B.5.1, ADA 5 , ADA 27
-, Ausnahmen bei B.11.3, ADA 46
- file B.11.1.1, B.11.2.1, ADA 22
- für Festpunktzahlen B.11.2.2.5, ADA 54
- - ganzzahlige Werte B.11.2.2.3, ADA 54
- - Gleitpunktzahlen B.11.2.2.4, ADA 54
- - logische Werte B.11.2.2.2, ADA 54
- - Werte von Aufzählungstypen B.11.2.2.6, ADA 54
- - Zeichen und Zeichenketten B.11.2.2.1, ADA 54
-, Standardfunktion für B.11, ADA 55

-, Standardprozeduren für B.11, ADA 54
Auslösen einer Ausnahme B.4.12, B.7.3, ADA 23
Ausnahme(-n) B.2, B.4.12, B.7
-, Auslösen einer B.7.3, ADA 23
- bei Ein- bzw. Ausgabe B.11.3, ADA 46
- behandler B.7.4
- behandlung B.7
- - teil B.7.4, ADA 46
- verbreitung B.7.5
- vereinbarung B.2.7, B.7.2, ADA 18
-, vordefinierte B.7.1, ADA 46
Ausprägung von Schemata B.6.2
Auswahl B.4.8, ADA 24
- anweisung B.4.8, ADA 24
- ausdruck B.2.1.1.2,B.4.8, ADA 21, ADA 24
- marke B.2.1.2.2, ADA 21
- typ B.4.8
Auswertung eines Ausdruckes B.3

B

Basistyp B.2.1, ADA 20
bedingte Anweisung B.4.4, ADA 24
Bezeichner A.2.2.1
Bibliothekseinheit B.1, B.10.1
Block B.2, B.4.9, ADA 24
- name B.4.9, ADA 24
- schachtelung B.10
boolean B.2.1.1.1.2, ADA 22
- Konstante → Konstante ADA 36

C

case-Anweisung B.4.8, ADA 24
character B.2.1.1.1.3, ADA 22
- Konstante → Konstante ADA 37
Compilerinstruktionen B.8, ADA 50, ADA 52

D

Darstellung(-s) B.8.2
-, angabe B.2, B.2.15, B.8.2, ADA 50
-, Aufzählungstyp- B.8.2.2, ADA 50
- höherer Sprachen A.2
-, Record- B.8.2.3, ADA 51
Datei (→ File) B.11
Daten
- objekte B.2.1
- satzliste B.2.1.2.2, ADA 21
- typen B.1.2
delay-Anweisung B.9.4, ADA 23
Diagramme A.2.1.1
diskreter Typ B.2.1.1.1, B.2.1.2.1, ADA 5
Diskrimmiante B.2, B.2.1.2.2, ADA 21, ADA 22
Direktive B.8.1, ADA 52
direkt sichtbar B.10.3
Disjunktion B.3.1, ADA 61
Durchlauf, durchlaufen A.2.2.1
duration B.2.1.1.2.2, B.10.1.3
dynamisches Array B.2.1.2.1
- variable → Zeiger

E

Eingabe B.11
- anweisung B.4.3, B.11, ADA 54
- argument B.1.1, B.5.1, B.6.1, ADA 5, ADA 27

-, Ausnahme bei - B.11.3, ADA 46
- file B.11.2.1, B.11.2.1, ADA 22
- für Festpunktzahlen B.11.2.2.5, ADA 54
- - ganzzahlige Werte B.11.2.2.3,ADA 54
- - Gleitpunktzahlen B.11.2.2.4,ADA 54
- - logische Werte B.11.2.2.2, ADA 54
- - Werte von Aufzählungstypen B.1.2.2.6, ADA 54
- - Zeichen und Zeichenketten B.11.2.2.1, ADA 54
-, Standardfunktionen für B.11, ADA 55
-, Standardprozeduren für B.11, ADA 54
Ein-/Ausgabe (→ Eingabe, → Ausgabe)
- argument B.5.1, B.6.1, ADA 5, ADA 27
- file B.11.1.1, ADA 22
Eingang(s) B.2, ADA 16
- aufruf B.9.2.2, ADA 23
- aufruf, bedinger B.9.3, ADA 24
- aufruf, verzögerter B.9.4, ADA 24
-, indizierter B.2, ADA 16
- vereinbarung B.9.2.1, ADA 16
- warteschlange B.9.2.4
entry-Vereinbarung B.9.2.1, ADA 16
Erzeugung
- Paket B.6.2, ADA 11
- Unterprogramm B.6.2, ADA 8
exit - Anweisung B.4.5, ADA 23
Exponentiation B.3.4, B.3.5, ADA 30

F

Feld → Array
fester Teil B.2.1.2.2, ADA 21
Festpunkt
- ausdruck B.3.5.2, ADA 28
- Konstante → Konstante ADA 40
- typ B.2.1.1.2.2, ADA 20, ADA 22

File(-s) B.11.1
-, aktuelle Länge eines - B.11.1.1
-, - Leseposition eines - B.11.1.1
-, - Schreibposition eines - B.11.1.1
-, Ausgabe - B.11.1.1, B.11.2.1, ADA 22
-, Ausdruck für B.11.4
-, Endposition eines - B.11.1.1
- index B.2.1.1.1.4, ADA 22
-, Text- B.11.2
-, typ B.11.1.1, ADA 22
-, verwaltung B.11.1.1
- - prozeduren B.11.1.1, ADA 53
float B.2.1.1.2.1, ADA 22
for-Klausel B.4.5, ADA 25
formale(-s)
- Argument(-Liste) B.1.1, B.5, ADA 27
- Schemaargument(-Liste) B.6.1, ADA 5
Funktion(-s)(→ Unterprogramm) B.5, ADA 7
- aufruf B.5.4, ADA 31

G

ganzzahlige(r)
- Division B.3.4
- Rest B.3.4
- Konstante ADA 36
generische Programmeinheit (→ Schema) B.6
geschützter Typ B.2.1.6, ADA 10, ADA 20
gleichrangig B.3
Gleitpunkt
- ausdruck B.3.5 ADA 28
- konstante → Konstante ADA 40
- typ B.2.1.1.2.1, ADA 20, ADA 22
globale Größe B.10.5
goto-Anweisung B.4.6, ADA 23

Grammatik A.2.2
Größe B.2
-, globale B.10.5
-, lokale B.10.5
Gültigkeitsbereich eines(r) B.10.2, B.10.5
- Diskriminante
- Eingangs
- formalen Argumentes
- getrennt übersetzten Paketes
- getrennt übersetzten Unterprogrammes
- Größe eines Programmes
- Literals eines Aufzähltyps
- Recordkomponente
- Schleifenparameters (Laufvariable)

I

I Konstante ADA 39
implizit vereinbart B.2
Index B.2.1.2.1
- bereich B.2.1.2.1, B.2.1.1.1.4, ADA 20
- bereichstyp B.2.1.1.1.4, ADA 22
- grenze B.2.1.2.1
- typ → Komponententyp B.2.1.1.1.4
Initialisierung B.2.1.2.1, B.2.1.2.2, B.2.2, ADA 18
input-output Paketschema B.11.1
integer B.2.1.1.1.4 ADA 22
- Konstante → Konstante ADA 39
-, universal_ B.2.1.1.1.4
Iterationsklausel B.4.5, ADA 25

K

kollektiv A.2.2.2
Kommentar B.1

kompatibel B.1.2, B.2.1.4, B.2.1.5
Komponente B.2, B.2.1.5
-, typ B.2.1.2.1, ADA 20
Konjunktion B.3.1, ADA 61
Konkatenation B.3.6.1, B.3.6.3, ADA 61
Konstante(n) B.1.2, B.2, B.2.1.
-, Vereinbarung von B.2.2, ADA 18, ADA 20
Kontextangabe B.10.4, ADA 2
Kurzschlußformen B.3, B.3.1

L

Längenangabe B.8.2.1, ADA 50
Leerzeichen B.1
limitierter Typ ADA 20
Linie, durchgezogen, gepunktet A.2.2.1
logisch(e,er)
- Ausdruck B.3.1, ADA 28
- Konstante ADA 36
lokale Größe B.10.4
long_float B.2.1.1.2.4, ADA 22
long_integer B.2.1.1.1.4, ADA 22
low_level_io, Paket B.11.3

M

Marken B.2, B.4.6, ADA 23
mehrschichtig A.2.2.2
Mitgliedschaft B.3.1, ADA
Modell
- intervall B.3.1
- zahl B.2.1.1.2, B.3.1
Modularität B.10
modulo-Operator B.3.4, ADA 30, ADA 61
Muttereinheit B.1.4 ADA 1, ADA 17

N

Nachfolgeanweisung B.1
Namen B.1, B.2, ADA 65
Namenskonflikt B.10.5
natural B.2.1.1.1.4, ADA 22
Negation B.3.1, ADA 30
new B.2.1.3, ADA 31

O

Operator (→ Unterprogramm) B.3, B.5, ADA 61
- aufruf B.5.5, ADA 28, ADA 29, ADA 30
Ordnung von Typen B.2.1.1
others-Zweig B.4.8, ADA 24

P

Paket
- erzeugung B.6.2, ADA 11
- low_level_io B.11.3
- schema input_output B.11.1
- text_io B.11.2
Parallelverarbeitung → Prozesse B.1.3
Parameter B.2
Pragma(-s) B.8.1
- für Speicherplatzaufteilung B.8.1
- - Festlegung der Objektmaschine B.8.1
- - Speicherverwaltung B.8.1
- - Aufruf von Unterprogrammen B.8.1
- - Optimierung B.8.1
- - Priorität von Prozessen B.8.1
- - Programmtext B.8.1
- - Unterdrücken von Tests B.8.1
Pragmatik A.2, A.2.4
Priorität eines Operators → Rangfolge

Priorität eines Prozesses B.9.6
priority B.2.1.1.1.4, B.9
Programm B.1, ADA 1
- aufbau B.1, ADA 1
- bibliothek B.10.1
- struktur B.10
Prozedur (→ Unterprogramm) B.5
- aufruf B.5.3, ADA 23
Prozeß B.1, B.1.3, B.2, B.9, ADA 1, ADA 14, ADA 15
- abhängigkeit B.9
-, aktiver B.9.2
- aktivierung B.9.1
- handhabung B.9
-, passiver B.9.2
- rendezvous B.9.2
- rendezvous, bedingtes B.9.3
- rendezvous, verzögertes B.9.4
- rumpf B.2.6, ADA 14
- spezifikation B.2.6, ADA 15
- synchronisierung B.9.2.4
- terminierung B.9.5
- typ B.9
- typvereinbarung B.2
- vereinbarung B.2.6, ADA 14, ADA 15

R

range B.2.1.1.1.4, B.2.1.2.1, ADA 33, ADA 47
Rangfolge B.3
raise-Anweisung B.7.3, ADA 23
Record B.2.1.2.1
- ausdruck B.3.7
- darstellung B.8.2.3, ADA 51
- konstante ADA 44
- typ B.2.2.2, ADA 20, ADA 21

Rendezvous von Prozessen B.9.2, B.9.2.4
-, bedingtes B.9.3
-, verzögertes B.9.4
reeller Typ B.2.1.1.2, ADA 22
referierter Typ B.2.1.3, ADA 20
Reihenfolge B.2
renames → Umbennung
Reservierter Name B.1, D.3
Resultattyp B.1.1, ADA 7
return-Anweisung B.4.7, B.5.2, ADA 23
Rücksprung, Unterprogramm- B.5.2, ADA 23
Rumpf, B.1, B.2, ADA 1, 6, 9, 14,
- stumpf B.2, B.10.1, ADA 17
- Unterprogramm- B.5, ADA 6
Rundung B.2.1.1.2

S

Schablone (→ Schema) B.6
Schema B.6
- argument, aktuelles B.2.5, B.6.3, ADA 13
- argumentliste, aktuelle B.6.3, ADA 12
- argument, formales B.6.1, ADA 5
- argumentliste, formale B.6.1, ADA 5
- ausprägung B.6.2
- Paketschema input_output B.11.1
- vereinbarung B.2.5, B.6.1, ADA 3, ADA 4
Schleife B.2, B.4.5
Schleifenindex B.4.5
Schreiben eines Programmes B.1
Schrittweite ADA 20
select-Anweisung B.9.3, ADA 24
selectiv A.2.2.2
Semantik A.2, A.2.3
short_float B.2.1.1.2.1, ADA 22

short_integer B.2.1.1.1.4, ADA 22
Sichtbarkeit eines(r) B.10.3
- Diskriminante
- Eingangs
- formalen Argumentes
- getrennt übersetzten Paketes
- getrennt übersetzten Unterprogrammes
- Größe eines Programmes
- Literals eines Aufzähltyps
- Recordkomponente
- Schleifenparameters (Laufvariable)
skalarer Typ B.2.1.1
slice B.2.2
Speicherbelegungsfunktion B.3.8, ADA 31
Speicherfreigabe, unkontrollierte B.8.3, ADA 8
Spezifikation, Unterprogramm- B.5.1, ADA 7
Sprunganweisung B.4.6, ADA 23
Sprung aus der Schleife B.4.5, ADA 23
Standardfunktionen
- - Ein- bzw. Ausgabe B.11, ADA 54
Standardname B.2
Standardprozeduren
- - Ein- bzw. Ausgabe
- - Fileverwaltung B.11, ADA 53
String
- Ausdruck → Ausdruck für Zeichenketten
- typ B.2.1.2.1.1, ADA 22
- konstante → Konstante ADA 42
strukturierte Anweisung ADA 24
Synchronisierung B.1.3, B.4.11, B.9.2.4
Symbol A.2.1, A.2.2.1
Syntax A.2, A.2.2
Syntaxdiagramm A.2.2.1

T

Teilfeld B.2.2
terminate-Alternative B.9.5
terminate-Anweisung B.9.5
Terminierung eines Prozesses B.4.11, B.9.5, ADA 23
-, abnormale B.9.5
Textfile(-s)(→ File) B.11.2
-, Spaltenposition eines B.11.2.1
- typ B.11.2.1
- verwaltung B.11.2.1
-, Zeilenende eines B.11.2.1
-, Zeilenlänge eines B.11.2.1
-, Zeilenposition eines B.11.2.1
text_io, Paket B.11.2
Trennzeichen B.1
τ-Syntaxdiagramm A.2.2.2
Typ(-en) B.2, ADA 22
-, Array B.2.1.2.1, ADA 20, ADA 47
-, Aufzählungs- B.2.1.1.1.1
-, diskreter B.2.1.1.1
-, Festpunkt- B.2.1.1.2.2, ADA 20, ADA 22
-, File- B.11.1.1, ADA 22
-, Gleitpunkt- B.2.1.1.2.1, ADA 20, ADA 22
- konvertierung B.3.10, ADA 31
-, reeller B.2.1.1.2, ADA 22
-, Record B.2.1.2.7, ADA 20, ADA 21
-, skalarer B.2.1, B.2.1.1
-, String B.2.1.2.1.1, ADA 22
-, strukturierter B.2.1, B.2.1.2
-, Textfile- B.11.2.1
-, Übersicht über die B.2.1
- umwandlung, unkontrollierte B.8.3, ADA 8
-, Vereinbarung von B.2.1, ADA 18
-, Zeiger- B.2.1, B.2.1.3

U

Überdecken von Namen B.10.5
Überladen
- von Aufzählungsliteralen B.2.1.1.1.1
- von Namen B.10.5
- von Operatoren B.3.9
- von Unterprogrammen B.5.7
Übersetzungseinheiten B.1, B.10.1, ADA 1
Übersetzung eines Programmes B.10.6
Umbenennung B.28 ADA 18
universal_integer → integer
Unterbereich B.2.1.1
Untereinheit B.1, B.1.4, B.10.1, ADA 1
Unterprogramm(-en)
(→ Funktion,→ Operator,→ Prozedur) B.1,B.2,B.5, ADA1,3,6,7
- erzeugung B.6.2, ADA 8
- rumpf B.5, ADA 6
- rücksprung B.4.7, B.5.2, ADA 23
- schema B.1, B.2.5, B.6
- spezifikation B.1.1, B.5.1, ADA 7
- Überladen von B.5.7
- vereinbarung B.2.3, B.5.1, ADA 6, ADA 7
Untertyp B.2.1.4, ADA 18, ADA 20
use-Klausel B.2, B.10.4, ADA 2, ADA 18

V

Variable B.2
-, Vereinbarung von B.2.2, ADA 18
Variantenteil B.2.1.2.2, ADA 21
Verarbeitung
- eines Unterprogrammes B.1.1
- eines Paketes B.2.1
- eines Prozesses B.1.3
- einer Bibliothekseinheit B.10.6

Verbreitung einer Ausnahme B.7.5
Vereinbarung B.2, ADA 17
-, Ausnahme - B.7.2, ADA 18
-, Eingangs- B.2.1, ADA 16
-, implizite B.2
-, Schema- B.6.1, ADA 3, ADA 4
-, Unterprogramm- B.5.1, ADA 6, ADA 7
Vergleich(-s) B.3.1, ADA 29
- operator B.3.1, ADA 61
verträglich B.2.1
Verweis → Zeiger
Verzweigung B.4.4, B.4.8, ADA 24
Verzögerung(-s)
- anweisung B.9.4, ADA 23
- zeit B.9.4, ADA 24
vordefinierte Ausnahme B.7.1, ADA 46
Vorgänger eines Prozesses B.9
vorzeitige (abnormale) Terminierung B.9.5, ADA 23
verzögertes Rendezvous B.9.4

W

Wertebereich B.2.1.1, B.2.1.5
Wertzuweisung B.4.2, ADA 23
while-Klausel B.4.5, ADA 25
Wiederholungsanweisung B.4.5, ADA 24
with-Klausel B.10.4, ADA 2
Wortsymbol A.2.2.1, B.1, D.2

Z

Zunamen B.1, ADA 58
Zeichen B.2.1.1.1.3
- konstante ADA 37
Zeichenkettenkonstante ADA 42

Zeiger Ausdruck B.3.8
- Konstante ADA 46
- Typ B.2.1.3 ADA 20, ADA 22
Zeilenendezeichen B.1
Zugriff → Zeiger
Zuname B.10.3, B.10.4, B.10.5, ADA 58
- eines Eingangs B.9.2.2
- einer Paketgröße B.10.4

E. LITERATUR

ADA

[1] ADA Reference Manual, Proposed Standard Document. United States Department of Defense, Springer, New York (1980/81)

[2] Barnes, J.G.P.: Programming in ADA. Addison Wesley, London (1982)

[3] Barth, G.: Die Programmiersprache ADA. Jahrbuch Überblicke Mathematik, S. 141-157, Bibliographisches Institut, Mannheim (1982)

[4] De Remer,F.e.a.: ADA Syntax Chart. Sigplan Notices 16,9 S. 48-60 (1981)

[5] Goos, G.: ADA: Zweck, Entwicklung und Zukunft einer Programmiersprache. Angewandte Informatik 24,2, S. 80-89 (1982)

[6] Harwood, W.T.: PASCAL to ADA: Upwards Compatibility. SWURCC Microprocessor Software Unit Report Series 2, University of Bath (März 1981)

[7] Hibbard, P.e.a.: Studies in ADA Style. Springer, New York/Heidelberg (1981)

[8] Hoare, C.A.R.: The Emperor's Old Clothes. CACM 24,2, S. 75-83 (1981)

[9] Ledgard, H.: ADA, an Introduction. ADA Reference Manual (July 1980). Springer, Heidelberg (1981)

[10] Nissen, J.C.D. e.a.: ADA-Europe Guidelines for the portability of ADA programs. NPL Report DNACS 52/81, Teddington (1981)

[11] Pyle, I.C.: Die Programmiersprache ADA. Hanser, München (1981)

[12] Rationale for the Design of the ADA Programming Language. Sigplan Notices 14,6 (1979)

[13] Wegner, P.: Programming with ADA, Prentice-Hall, Englewood Cliffs (1980)

[14] Wichmann, B.A.: Tutorial Material on the real datatypes in ADA. Final Technical Report, NPL, Teddington (1981)

[15] Winkler, J.F.H.: ADA: Die neuen Konzepte. Elektron. Rechnenanlagen 24, S. 175-186 (1982)

SONSTIGE:

[16] Alefeld, G., Herzberger, J.: Einführung in die Intervallrechnung (Reihe Informatik, Bd. 12), Bibliographisches Institut, Mannheim (1974)

[17] Bohlender, G., Grüner, K., Kaucher, E., Klatte, R., Krämer, W., Kulisch, U., Miranker, W.L., Rump, S.M., Ullrich, Ch., Wolff v. Gudenberg, J.: PASCAL-SC: A Pascal for Contemporary Scientific Computation, IBM Research Report RC 9009, Yorktown (1981)

[18] Bohlender, G., Grüner, K., Kaucher, E., Klatte, R., Krämer, W., Kulisch, U., Miranker, W.L., Rump, S.M., Ullrich, W., Wolff v. Gudenberg, J., Böhm, H.: MATRIX PASCAL. Erscheint als IBM Research Report (1982)

[19] Bohlender, G., Kaucher, E., Klatte, R., Kulisch, U., Miranker, W.L., Ullrich, Ch., Wolff v. Gudenberg,J.: FORTRAN for Contemporary Numerical Computation, IBM Report, New York (August 1980)

[20] Berkely, E.C., Bobrow, D.G.: The Programming Language LISP - Operations and Applications. M.I.T. Press, Cambridge, Mass. (1966)

[21] Claus, V.: Einführung in die Informatik, Teubner, Stuttgart (1975)

[22] Dahl, J., Nygaard, K.: SIMULA - an ALGOL - Based Simulation Language. CACM 9, 671-678 (1966)

[23] DIN 66001, Informationsverarbeitung: Sinnbilder für Datenfluß- und Programmablaufpläne. Beuth, Berlin (1966)

[24] Hahn, R.: Höhere Programmiersprachen im Vergleich. Akad. Verlagsgesellschaft Wiesbaden (1981)

[25] Jordan, W., Urban, H.: Strukturierte Programmierung. Springer, Berlin (1978)

[26] Kaucher, E., Klatte, R., Ullrich, Ch.: Neuere Methoden zur Beschreibung von Programmiersprachen, Jahrbuch Überblicke Mathematik, Bibliographisches Institut, Mannheim (1978)

[27] Kaucher, E., Klatte, R., Ullrich, Ch.: Benutzerfreundliche Darstellung der Syntax von PASCAL durch Syntaxdiagramme. Applied Computer Science, Berichte zur praktischen Informatik 11, 43-62, Hanser, München (1978)

[28] Kaucher, E., Klatte, R., Ullrich, Ch.: Höhere Programmiersprachen ALGOL, FORTRAN, PASCAL in einheitlicher und übersichtlicher Darstellung (Reihe In-

formatik, Bd. 24). Bibliographisches Institut, Mannheim (1980)

[29] Kaucher, E., Klatte, R., Ullrich, Ch.: Programmiersprachen im Griff. Band 1: FORTRAN. Bibliographisches Institut, Mannheim (1980)

[30] Kaucher, E., Klatte, R., Ullrich, Ch.: Programmiersprachen im Griff. Band 2: PASCAL. Bibliographisches Institut, Mannheim (1981)

[31] Kaucher, E., Klatte, R., Ullrich, Ch.: Programmiersprachen im Griff. Band 3: BASIC. Bibliographisches Institut, Mannheim (1981)

[32] Krauß, F.: Programmiertechnik. Vogel, Würzburg (1971)

[33] Kulisch, U.: Grundlagen des numerischen Rechnens (Reihe Informatik, 19). Bibliographisches Institut, Mannheim (1976)

[34] Kulisch, U., Miranker, W.L.: Computer Arithmetic in Theory and Practice. Academic Press, New York (1981)

[35] Kulisch, U., Miranker, W.L. (Hrsg.): Proceedings of the "IBM Symposium on: A New Approach to Scientific Computation". Erscheint in Academic Press, New York (1983)

[36] Kulisch, U., Ullrich, Ch. (Hrsg.): Wissenschaftliches Rechnen und Programmiersprachen, Berichte des German Chapter of the ACM, 10, Teubner (1982)

[37] Maurer, H.: Theoretische Grundlagen der Programmiersprachen. Bibliographisches Institut, Mannheim (1969)

[38] Mell, W.-D., Preus, P., Sandner, P.: Einführung in die Programmiersprachen PL/1. Bibliographisches Institut, Mannheim (1974)

[39] Mickel, K.-P.: Einführung in die Programmiersprache COBOL. Bibliographisches Institut, Mannheim (1975)

[40] Rohlfing, M.: SIMULA. Bibliographisches Institut, Mannheim (1973)

[41] Sammet, J.E.: Roster of Programming Languages for 1974-75. CACM 19, 655-669 (1976)

[42] Schneider, H.J.: Problemorientierte Programmiersprachen, Teubner, Stuttgart

[43] van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Kester, C.H.A.: Report von the Algorithmic Language ALGOL 68, Numer. Math. 14, 79-218 (1969)

[44] v. Neumann, John: Entwicklung und Ausnutzung neuerer mathematischer Maschinen. Arbeitsgemeinschaft für

Forschung des Landes Nordrhein-Westfalen, Heft 45, Westdeutscher Verlag Köln und Opladen (1955)

[45] Wilkinson, J.H.: Rundungsfehler. Springer, Berlin (1969)

Die wissenschaftlichen Veröffentlichungen aus dem Bibliographischen Institut

B. I.-Hochschultaschenbücher, Einzelwerke und Reihen

Mathematik, Informatik, Physik, Astronomie, Philosophie, Chemie, Medizin, Ingenieurwissenschaften, Gesellschaft/Recht/Wirtschaft, Geowissenschaften

B·I·

Wissenschaftsverlag
Bibliographisches Institut

Inhaltsverzeichnis

Sachgebiete

Mathematik 2
Reihen:
Jahrbuch Überblicke Mathematik..... 9
Methoden und Verfahren der mathematischen Physik 9
Informatik 10
Reihe: Informatik 11
Physik 12
Astronomie 15
Philosophie 15
Reihe: Grundlagen der exakten Naturwissenschaften 16
Chemie 16
Medizin 17
Ingenieurwissenschaften 17
Reihe: Theoretische und experimentelle Methoden der Regelungstechnik 19
Gesellschaft/Recht/Wirtschaft .. 19
Geographie, Geologie 20
B.I.-Hochschulatlanten 20

Zeichenerklärung
HTB = B.I.-Hochschultaschenbücher.
Wv = B.I.-Wissenschaftsverlag (Einzelwerke und Reihen).
M.F.O. = Mathematische Forschungsberichte Oberwolfach.

Stand: Juni 1982

Mathematik

Aitken, A. C.
Determinanten und Matrizen
142 S. mit Abb. 1969. (HTB 293)

Andrié, M./P. Meier
Analysis
Eine anwendungsbezogene Einführung
257 S. 1981. (HTB 602)

Andrié, M./P. Meier
Lineare Algebra und analytische Geometrie. Eine anwendungsbezogene Einführung
243 S. 1977. (HTB 84)

Artmann, B./W. Peterhänsel/ E. Sachs
Beispiele und Aufgaben zur linearen Algebra
150 S. 1978. (HTB 783)

Aumann, G.
Höhere Mathematik
Band I: Reelle Zahlen, Analytische Geometrie, Differential- und Integralrechnung. 243 S. mit Abb. 1970. (HTB 717)
Band II: Lineare Algebra, Funktionen mehrerer Veränderlicher. 170 S. mit Abb. 1970. (HTB 718)
Band III: Differentialgleichungen. 174 S. 1971. (HTB 761)

Bandelow, Ch.
Einführung in die Wahrscheinlichkeitstheorie
206 S. 1981. (HTB 798)

Barner, M./W. Schwarz (Hrsg.)
Zahlentheorie
235 S. 1971. (M. F. O. 5)

Behrens, E.-A.
Ringtheorie
405 S. 1975. Wv.

Beutelspacher, A.
Einführung in die endliche Geometrie I. Blockpläne
247 S. 1982. Wv.

Böhmer, K./G. Meinardus/ W. Schempp (Hrsg.)
Spline-Funktionen. Vorträge und Aufsätze
415 S. 1974. Wv.

Brandt, S.
Datenanalyse. Mit statistischen Methoden und Computerprogrammen
464 S. 2., erw. Aufl. 1981. Wv.

Brauner, H.
Geometrie projektiver Räume
Band I: Projektive Ebenen, projektive Räume. 235 S. 1976. Wv.
Band II: Beziehungen zwischen projektiver Geometrie und linearer Algebra. 258 S. 1976. Wv.

Brosowski, B.
Nichtlineare Tschebyscheff-Approximation
153 S. 1968. (HTB 808)

Brosowski, B./R. Kreß
Einführung in die numerische Mathematik
Band I: Gleichungssysteme, Approximatiostheorie. 223 S. 1975. (HTB 202)
Teil II: Interpolation, numerische Integration, Optimierungsaufgaben. 124 S. 1976. (HTB 211)

Brunner, G.
Homologische Algebra
213 S. 1973. Wv.

Cartan, H.
Differentialformen
250 S. 1974. Wv.

Cartan, H.
Differentialrechnung
236 S. 1974. Wv.

Cartan, H.
Elementare Theorie der analytischen Funktionen einer oder mehrerer komplexen Veränderlichen
236 S. mit Abb. 1966. (HTB 112)

Cigler, J./H.-C. Reichel
Topologie. Eine Grundvorlesung
257 S. 1978. (HTB 121)

Degen, W./K. Böhmer
Gelöste Aufgaben zur Differential- und Integralrechnung
Band I: Eine reelle Veränderliche.
254 S. 1971. (HTB 762)

Dombrowski, P.
Differentialrechnung I und Abriß der linearen Algebra
271 S. mit Abb. 1970. (HTB 743)

Egle, K.
Graphen und Präordnungen
207 S. 2. Aufl. 1981. Wv.

Eisenack, G./C. Fenske
Fixpunkttheorie
258 S. 1978. Wv.

Elsgolc, L. E.
Variationsrechnung
157 S. mit Abb. 1970. (HTB 431)

Eltermann, H.
Grundlagen der praktischen Matrizenrechnung
128 S. mit Abb. 1969. (HTB 434)

Ernè, M.
Einführung in die Ordnungstheorie
Etwa 250 S. 1982. Wv.

Erwe, F.
Differential- und Integralrechnung
Band I: Differentialrechnung.
364 S. mit Abb. 1962. (HTB 30)
Band II: Integralrechnung.
197 S. mit 50 Abb. 1973. (HTB 31)

Erwe, F.
Gewöhnliche Differentialgleichungen
152 S. mit 11 Abb. 1964. (HTB 19)

Erwe, F.
Reelle Analysis
(Reihe: Mathematik für Physiker, Band 5)
360 S. 1978. Wv.

Erwe, F./E. Peschl
Partielle Differentialgleichungen erster Ordnung
133 S. 1973. (HTB 87)

Ewald, G.
Probleme der geometrischen Analysis
160 S. mit ca. 110 Figuren. 1982. Wv.

Felscher, W.
Naive Mengen und abstrakte Zahlen
Band I: Die Anfänge der Mengenlehre und die natürlichen Zahlen.
260 S.1978. Wv.
Band II: Die Struktur der algebraischen und der reellen Zahlen.
222 S. 1978. Wv.
Band III: Transfinite Methoden.
272 S. 1979. Wv.

Fuchssteiner, B./D. Laugwitz
Funktionalanalysis
(Reihe: Mathematik für Physiker, Band 9)
219 S. 1974. Wv.

Gericke, H.
Geschichte des Zahlbegriffs
163 S. mit Abb. 1970. (HTB 172)

Goffmann, C.
Reelle Funktionen
331 S. Aus dem Englischen. 1976. Wv.

Gröbner, W.
Algebraische Geometrie
Band I: Allgemeine Theorie der kommutativen Ringe und Körper.
193 S. 1968. (HTB 273)

Gröbner, W.
Differentialgleichungen I.
Gewöhnliche Differentialgleichungen
(Reihe: Mathematik für Physiker, Band 6)
188 S. 1977. Wv.

Gröbner, W.
Differentialgleichungen II.
Partielle Differentialgleichungen
(Reihe: Mathematik für Physiker, Band 7)
157 S. 1977. Wv.

Gröbner, W./H. Knapp
Contributions to the Method of Lie Series
In englischer Sprache.
265 S. 1967. (HTB 802)

Grotemeyer, K. P./E. Letzner/ R. Reinhardt
Topologie
187 S. mit Abb. 1969. (HTB 836)

Gumm, H. P./W. Poguntke
Boolesche Algebra
95 S. 1981. (HTB 604)

Hämmerlin, G.
Numerische Mathematik I
Band I: Approximation, Interpolation, Numerische Quadratur, Gleichungssysteme.
199 S. 2., überarbeitete Aufl. 1978. (HTB 498)

Hasse, H./P. Roquette (Hrsg.)
Algebraische Zahlentheorie
272 S. 1966. (M. F. O. 2)

Heidler, K./H. Hermes/ F.-K. Mahn
Rekursive Funktionen
248 S. 1977. Wv.

Heil, E.
Differentialformen
207 S. 1974. Wv.

Hein, O.
Graphentheorie für Anwender
141 S. 1977. (HTB 83)

Hein, O.
Statistische Verfahren der Ingenieurpraxis
197 S. Mit 5 Tabellen, 6 Diagrammen, 43 Beispielen. 1978. (HTB 119)

Hellwig, G.
Höhere Mathematik
Band I/1. Teil: Zahlen, Funktionen, Differential- und Integralrechnung einer unabhängigen Variablen.
284, IX S. 1971. (HTB 553)
Band I/2. Teil: Theorie der Konvergenz Ergänzungen zur Integralrechnung, das Stieltjes-Integral. 137 S. 1972. (HTB 560)

Hengst, M.
Einführung in die mathematische Statistik und ihre Anwendung
259 S. mit Abb. 1967. (HTB 42)

Henze, E.
Einführungen in die Maßtheorie
235 S. 1971. (HTB 505)

Heyer, H.
Einführung in die Theorie Markoffscher Prozesse
253 S. 1979. Wv.

Hirzebruch, F./W. Scharlau
Einführung in die Funktionalanalysis
178 S. 1971. (HTB 296)

Hlawka, E.
Theorie der Gleichverteilung
152 S. 1979. Wv.

Holmann, H./H. Rummler
Alternierende Differentialformen
257 S. 2., durchgesehene Aufl. 1981. Wv.

Horvath, H.
Rechenmethoden und ihre Anwendung in Physik und Chemie
142 S. 1977. (HTB 78)

Hoschek, J.
Liniengeometrie
VI, 263 S. mit Abb. 1971. (HTB 733)

Hoschek, J./G. Spreitzer
Aufgaben zur darstellenden Geometrie
229 S. mit Abb. 1974. Wv.

Ince, E. L.
Die Integration gewöhnlicher Differentialgleichungen
180 S. Aus dem Englischen. 1965. (HTB 67)

Joachim, E.
Einführung in die Algebra
168 S. 2. Aufl. 1980. (HTB 138)

Jordan-Engeln, G./F. Reutter
Formelsammlung zur Numerischen Mathematik mit Standard Fortran-Programmen
424 S. 3., überarb. und erweiterte Aufl. 1981. (HTB 106)

Jordan-Engeln, G./F. Reutter
Numerische Mathematik für Ingenieure
Etwa 400 S. 3., überarbeitete und erw. Aufl. 1982. (HTB 104)

Kaiser, R./G. Gottschalk
Elementare Tests zur Beurteilung von Meßdaten
68 S. 1972. (HTB 774)

Kießwetter, K.
Reelle Analysis einer Veränderlichen. Ein Lern- und Übungsbuch
316 S. 1975. (HTB 269)

Kießwetter, K./R. Rosenkranz
Lösungshilfen für Aufgaben zur reellen Analysis einer Veränderlichen
231 S. 1976. (HTB 270)

Klingbeil, E.
Tensorrechnung für Ingenieure
197 S. mit Abb. 1966. (HTB 197)

Klingbeil, E.
Variationsrechnung
332 S. 1977. Wv.

Klingenberg, W. (Hrsg.)
Differentialgeometrie im Großen
351 S. 1971. (M. F. O. 4)

Klingenberg, W./P. Klein
Lineare Algebra und analytische Geometrie
Band I: Grundbegriffe, Vektorräume. XII, 288 S. 1971. (HTB 748)
Band II: Determinanten, Matrizen, Euklidische und unitäre Vektorräume. XVIII, 404 S. 1972. (HTB 749)

Klingenberg, W./P. Klein
Lineare Algebra und analytische Geometrie. Übungen zu Band I–II
VIII, 172 S. 1973. (HTB 750)

Laugwitz, D.
Infinitesimalkalkül. Kontinuum und Zahlen. Eine elementare Einführung in die Nichtstandard-Analysis
187 S. 1978. Wv.

Laugwitz, D.
Ingenieurmathematik
Band I: Zahlen, analytische Geometrie, Funktionen.
158 S. mit 43 Abb. 1964. (HTB 59)
Band II: Differential- und Integralrechnung.
152 S. mit 43 Abb. 1964. (HTB 60)
Band III: Gewöhnliche Differentialgleichungen.
141 S. 1964. (HTB 61)
Band IV: Fourier-Reihen, verallgemeinerte Funktionen, mehrfache Integrale, Vektoranalysis, Differentialgeometrie, Matrizen, Elemente der Funktionalanalysis.
196 S. mit Abb. 1967. (HTB 62)
Band V: Komplexe Veränderliche.
158 S. mit Abb. 1965. (HTB 93)

Laugwitz, D./C. Schmieden
Aufgaben zur Ingenieurmathematik
182 S. 1966. (HTB 95)

Lebedew, N. N.
Spezielle Funktionen und ihre Anwendung
372 S. mit Abb. Aus dem Russischen. 1973. Wv.

Lidl, R./G. Pilz
Angewandte abstrakte Algebra
Band I: 249 S. 1982. Wv.
Band II: Etwa 240 S. 1982. Wv.

Lighthill, M. J.
Einführung in die Theorie der Fourieranalysis und der verallgemeinerten Funktionen
96 S. mit Abb. Aus dem Englischen. 1966. (HTB 139)

Lingenberg, R.
Grundlagen der Geometrie
224 S. mit 73 Abb. 3., durchgesehene Aufl. 1978. Wv.

Lingenberg, R.
Lineare Algebra
161 S. 1969. (HTB 828)

Lorenz, F.
Lineare Algebra
Band I: 233 S. 1982. (HTB 601)
Band II: Etwa 200 S. 1982. (HTB 605)

Lorenzen, P.
Metamathematik
175 S. 2. Aufl. 1980. Wv.

Lüneburg, H.
Galoisfelder, Kreisteilungskörper und Schieberegisterfolgen
143 S. 1979. Wv.

Lüneburg, H.
Vorlesungen über Analysis
467 S. 1981. Wv.

Lutz, D.
Topologische Gruppen
175 S. 1976. Wv.

Mainzer, K.
Geschichte der Geometrie
232 S. 1980. Wv.

Marsal, D.
Die numerische Lösung partieller Differentialgleichungen in Wissenschaft und Technik
XXVIII, 574 S. mit Abb. 1976. Wv.

Martensen, E.
Analysis.
Für Mathematiker, Physiker, Ingenieure
Band I: Grundlagen der Infinitesimalrechnung.
IX, 200 S. 2. Aufl. 1976. (HTB 832)
Band II: Aufbau der Infinitesimalrechnung.
VIII, 176 S. 2., neu bearbeitete Aufl. 1978. (HTB 833)
Band III: Gewöhnliche Differentialgleichungen.
IX, 237 S. mit 52 Abb. 2., neu bearbeitete Aufl. 1980. (HTB 834)

Meinardus, G.
Approximation in Theorie und Praxis. Ein Symposiumbericht
304 S. 1979. Wv.

Meinardus, G./G. Merz
Praktische Mathematik I.
Für Ingenieure, Mathematiker und Physiker
Band I: 346 S. 1979. Wv.
Band II: 427 S. 1982. Wv.

Meschkowski, H.
Einführung in die moderne Mathematik
214 S. mit 44 Abb. 3., verbesserte Aufl. 1971. (HTB 75)

Meschkowski, H.
Elementare Wahrscheinlichkeitsrechnung und Statistik
(Reihe: Mathematik für Physiker, Band 3)
188 S. 1972. Wv.

Meschkowski, H.
Funktionen
(Reihe: Mathematik für Physiker, Band 2)
179 S. mit 66 Abb. 1970. Wv.

Meschkowski, H.
Grundlagen der Euklidischen Geometrie
231 S. mit 145 Abb. 2., verbesserte Aufl. 1974. Wv.

Meschkowski, H.
Mathematik und Realität. Vorträge und Aufsätze.
184 S. 1979. Wv.

Meschkowski, H.
Mathematiker-Lexikon
342 S. 3., überarbeitete und ergänzte Aufl. 1980. Wv.

Meschkowski, H.
Mathematisches Begriffswörterbuch
315 S. mit Abb. 4. Aufl. 1976. (HTB 99)

Meschkowski, H.
Mehrsprachenwörterbuch mathematischer Begriffe
135 S. 1972. Wv.

Meschkowski, H.
Problemgeschichte der Mathematik I
206 S. 1979. Wv.

Meschkowski, H.
Problemgeschichte der Mathematik II
235 S. 1981. Wv.

Meschkowski, H.
Problemgeschichte der neueren Mathematik (1800–1950)
314 S. mit Abb. 1978. Wv.

Meschkowski, H.
Richtigkeit und Wahrheit in der Mathematik
219 S. 2., durchgesehene Aufl. 1978. Wv.

Meschkowski, H.
Unendliche Reihen
320 S. 2., verb. und erweiterte Aufl. 1982. Wv.

Meschkowski, H.
Ungelöste und unlösbare Probleme der Geometrie
204 S. 2., verb. und erweiterte Aufl. 1975. Wv.

Meschkowski, H.
Wahrscheinlichkeitsrechnung
233 S. mit Abb. 1968. (HTB 285)

Meschkowski, H.
Zahlen
(Reihe: Mathematik für Physiker, Band 1)
174 S. mit 37 Abb. 1970. Wv.

Meschkowski, H./I. Ahrens
Theorie der Punktmengen
183 S. mit Abb. 1974. Wv.

Niven, I./H. S. Zuckermann
Einführung in die Zahlentheorie
Band I: Teilbarkeit, Kongruenzen, quadratische Reziprozität u. a.
213 S. Aus dem Englischen. 1976. (HTB 46)
Band II: Kettenbrüche, algebraische Zahlen, die Partitionsfunktion u. a.
186 S. Aus dem Englischen. 1976. (HTB 47)

Noble, B.
Numerisches Rechnen
Band II: Differenzen, Integration und Differentialgleichungen.
246 S. Aus dem Englischen. 1973. (HTB 147)

Oberschelp, A.
Elementare Logik und Mengenlehre
Band I: Die formalen Sprachen, Logik.
254 S. 1974. (HTB 407)
Band II: Klassen, Relationen, Funktionen, Anfänge der Mengenlehre.
229 S. 1978. (HTB 408)

Peschl, E.
Differentialgeometrie
92 S. 1973. (HTB 80)

Peschl, E.
Funktionentheorie
Band I: 274 S. mit Abb. 1967. (HTB 131)

Poguntke, W./R. Wille
Testfragen zur Analysis I
117 S. 2. Aufl. 1980. (HTB 781)

Preuß, G.
Grundbegriffe der Kategorientheorie
105 S. 1975. (HTB 739)

Reiffen, H.-J./H. W. Trapp
Einführung in die Analysis
Band II: Theorie der analytischen und differenzierbaren Funktionen.
260 S. 1973. (HTB 786)

Rommelfanger, H.
Differenzen- und Differentialgleichungen
232 S. 1977. Wv.

Rothstein, W./K. Kopfermann
Funktionentheorie mehrerer komplexer Veränderlicher
256 S. 1982. Wv.

Rottmann, K.
Mathematische Formelsammlung
176 S. mit 39 Abb. 1962. (HTB 13)

Rottmann, K.
Siebenstellige dekadische Logarithmen
194 S. 1960. (HTB 17)

Rutsch, M.
Wahrscheinlichkeit I
350 S. mit Abb. 1974. Wv.

Rutsch, M./K.-H. Schriever
Wahrscheinlichkeit II
404 S. mit Abb. 1976. Wv.

Rutsch, M./K.-H. Schriever
Aufgaben zur Wahrscheinlichkeit
267 S. mit Abb. 1974. Wv.

Schick, K.
Lineare Optimierung
331 S. mit Abb. 1976. (HTB 64)

Schmidt, J.
Mengenlehre. Einführung in die axiomatische Mengenlehre
Band I: 245 S. mit Abb. 2., verb. und erweiterte Aufl. 1974. (HTB 56)

Schwabhäuser, W.
Modelltheorie
Band II: 123 S. 1972. (HTB 815)

Schwartz, L.
Mathematische Methoden der Physik
Band I: Summierbare Reihen, Lebesque-Integral, Distributionen, Faltung. Aus dem Französischen.
184 S. 1974. Wv.

Schwarz, W.
Einführung in die Siebmethoden der analytischen Zahlentheorie
215 S. 1974. Wv.

Spallek, K.
Kurven und Karten
272 S. 1980. Wv.

Tamaschke, O.
Permutationsstrukturen
276 S. 1969. (HTB 710)

Tamaschke, O.
Schur-Ringe
240 S. mit Abb. 1970. (HTB 735)

Teichmann, H.
Physikalische Anwendungen der Vektor- und Tensorrechnung
231 S. mit 64 Abb. 3. Aufl. 1975. (HTB 39)

Uhde, K.
Spezielle Funktionen der mathematischen Physik
Band I: Tafeln, Zylinderfunktionen.
267 S. 1964. (HTB 55)

Voigt, A./J. Wloka
Hilberträume und elliptische Differentialoperatoren
260 S. 1975. Wv.

Waerden, B. L. van der
Mathematik für Naturwissenschaftler
280 S. mit 167 Abb. 1975. (HTB 281)

Wagner, K.
Graphentheorie
220 S. mit Abb. 1970. (HTB 248)

Walter, R.
Differentialgeometrie
286 S. 1978. Wv.

Walter, W.
Einführung in die Theorie der Distributionen
VIII, 211 S. mit Abb. 1974. Wv.

Weizel, R./J. Weyland
Gewöhnliche Differentialgleichungen. Formelsammlung mit Lösungsmethoden und Lösungen
194 S. mit Abb. 1974. Wv.

Werner, H.
Einführung in die allgemeine Algebra
152 S. 1978. (HTB 120)

Werner, H. und I./P. Janßen/ H. Arndt
Probleme der praktischen Mathematik. Eine Einführung
Band I: mathematische Hilfsmittel, Fehlertheorie, Lösung von Gleichungssystemen u. a. 159 S. 2. Aufl. 1980. (HTB 134)
Band II: Interpolation, Approximation, numerische Differentiation und Integration, gewöhnliche Differentialgleichungen u. a. 169 S. 2. Aufl. 1980. (HTB 135)

Wollny, W.
Reguläre Parkettierung der euklidischen Ebene durch unbeschränkte Bereiche
316 S. mit Abb. 1970. (HTB 711)

Wunderlich, W.
Darstellende Geometrie
Band I: 187 S. mit Abb. 1966. (HTB 96)
Band II: 234 S. mit Abb. 1967. (HTB 133)

Reihe: Jahrbuch Überblicke Mathematik

Herausgegeben von Prof. Dr. S. D. Chatterji, Eidgen. Techn. Hochschule Lausanne, Prof. Dr. Istvan Fenyö, Techn. Universität Budapest, Prof. Dr. Benno Fuchssteiner, Gesamthochschule Paderborn, Prof. Dr. Ulrich Kulisch, Universität Karlsruhe, Prof. Dr. Detlef Laugwitz, Techn. Hochschule Darmstadt, Prof. Dr. Roman Liedl, Universität Innsbruck.

Die Jahrbücher 1975, 1977 und 1979 sind vergriffen.

Jahrbuch Überblicke Mathematik 1976. 204 S. mit Abb. 1976. Wv.

Jahrbuch Überblicke Mathematik 1978. 224 S. 1978. Wv.

Jahrbuch Überblicke Mathematik 1980. 214 S. 1980. Wv.

Jahrbuch Überblicke Mathematik 1981. 264 S. 1981. Wv.

Jahrbuch Überblicke Mathematik 1982. 218 S. 1982. Wv.

Reihe: Überblicke Mathematik

Herausgegeben von Prof. Dr. Detlef Laugwitz, Techn. Hochschule Darmstadt.

Band 1: 213 S. 1968. (HTB 161)
Band 2: 210 S. 1969. (HTB 232)
Band 3: 157 S. 1970. (HTB 247)
Band 4: 123 S. 1972. Wv.
Band 5: 186 S. 1972. Wv.
Band 6: 242 S. mit Abb. 1973. Wv.
Band 7: 265, II S. mit Abb. 1974. Wv.

Reihe: Methoden und Verfahren der mathematischen Physik

Herausgegeben von Prof. Dr. Bruno Brosowski, Universität Frankfurt und Prof. Dr. Erich Martensen, Universität Karlsruhe.

Band 1: 183 S. 1969. (HTB 720)
Band 2: 179 S. 1970. (HTB 721)
Band 3: 176 S. 1970. (HTB 722)
Band 4: 177 S. 1971. (HTB 723)
Band 5: 199 S. 1971. (HTB 724)
Band 6: 163 S. 1972. (HTB 725)
Band 7: 176 S. 1972. (HTB 726)
Band 8: 222 S. mit Abb. 1973. Wv.
Band 9: 201 S. mit Abb. 1973. Wv.
Band 10: 184 S. 1973. Wv.
Band 11: 190 S. mit Abb. 1974. Wv.
Band 12: 214 S. mit Abb. 1975. Mathematical Geodesy, Part. 1. Wv.
Band 13: 206 S. mit Abb. 1975. Mathematical Geodesy, Part. 2. Wv.
Band 14: 176 S. mit Abb. 1975. Mathematical Geodesy, Part. 3. Wv.
Band 15: 166 S. 1976. Wv.
Band 16: 180 S. 1976. Wv.

Informatik

Alefeld, G./J. Herzberger/ O. Mayer
Einführung in das Programmieren mit ALGOL 60
164 S. 1972. (HTB 777)

Bosse, W.
Einführung in das Programmieren mit ALGOL W
249 S. 1976. (HTB 784)

Breuer, H.
Algol-Fibel
120 S. mit Abb. 1973. (HTB 506)

Breuer, H.
Fortran-Fibel
85 S. mit Abb. 1969. (HTB 204)

Breuer, H.
PL/1-Fibel
106 S. 1973. (HTB 552)

Bruderer, H. E.
Nichtnumerische Datenverarbeitung
267 S. mit 60 Abb. 2. Aufl. 1980. Wv.

Dederichs, W.
APPLESOFT-BASIC
188 S. 1982. (HTB 603)

Dotzauer, E.
Einführung in APL
248 S. 1978. (HTB 753)

Haase, V./W. Stucky
BASIC
Programmieren für Anfänger
230 S. 1977. (HTB 744)

Händler, W./G. Nees (Hrsg.)
Rechnergestützte Aktivitäten CAD
176 S. 1980. Wv.

Hainer, K.
Numerische Algorithmen auf programmierbaren Taschenrechnern
263 S. 1980. (HTB 805)

Kaucher, E./R. Klatte/Ch. Ullrich
Programmiersprachen im Griff
Band 1: FORTRAN
310 S. 1980. (HTB 795)
Band 2: PASCAL
359 S. 1981. (HTB 796)
Band 3: BASIC
390 S. 1981. (HTB 797)

Lawson jr., H. W./E. J. Neuhold
Verstehen Sie Datenverarbeitung?
Etwa 250 S. 1982. Wv.

Mell, W.-D./P. Preuß/P. Sandner
Einführung in die Programmiersprache PL/1
304 S. 1974. (HTB 785)

Mickel, K.-P.
Einführung in die Programmiersprache COBOL
219 S. 2., verbesserte Aufl. 1980. (HTB 745)

Müller, K. H./I. Streker
FORTRAN.
Programmierungsanleitung
215 S. 2. Aufl. 1970. (HTB 804)

Rohlfing, H.
PASCAL. Eine Einführung
217 S. 1978. (HTB 756)

Rohlfing, H.
SIMULA
243 S. mit Abb. 1973. (HTB 747)

Schließmann, H.
Programmierung mit PL/1
206 S. 2., erweiterte Aufl. 1978. (HTB 740)

Weber, H./J. Grami
Numerische Verfahren für programmierbare Taschenrechner I
192 S. 1980. (HTB 803)

Zimmermann, G./J. Höffner
Elektrotechnische Grundlagen der Informatik II
Wechselstromlehre, Leitungen, analoge u. digitale Verarbeitung kontinuierlicher Signale.
194 S. mit Abb. 1974. (HTB 790)

Reihe: Informatik

Herausgegeben von Prof. Dr. Karl Heinz Böhling, Universität Bonn, Prof. Dr. Ulrich Kulisch, Universität Karlsruhe und Prof. Dr. Hermann Maurer, Technische Universität Graz.

Band 1:
Maurer, H.
Theoretische Grundlagen der Programmiersprachen. Theorie der Syntax
254 S. Unveränderter Neudruck 1977. Wv.

Band 2:
Heinhold, J./U. Kulisch
Analogrechnen
242 S. mit Abb. 1976. Wv.

Band 7:
Kameda, T./K. Weihrauch
Einführung in die Codierungstheorie Teil I: 218 S. 1973. Wv.

Band 8:
Reusch, B.
Lineare Automaten
149 S. mit Abb. 1969. (HTB 708)

Band 10:
Böhling, K. H./G. Dittrich
Endliche stochastische Automaten
138 S. 1972. (HTB 766)

Band 11:
Seegmüller, G.
Einführung in die Systemprogrammierung
480 S. mit 83 Abb. 1974. Wv.

Band 12:
Alefeld, G./J. Herzberger
Einführung in die Intervallrechnung
XIII, 398 S. mit Abb. 1974. Wv.

Band 13:
Duske, J./H. Jürgensen
Codierungstheorie
235 S. 1977. Wv.

Band 14:
Böhling, K. H./B. v. Braunmühl
Komplexität bei Turingmaschinen
324 S. 1974. Wv.

Band 15:
Peters, F. E.
Einführung in mathematische Methoden der Informatik
348 S. 1974. Wv.

Band 16:
Wedekind, H.
Datenbanksysteme I
325 S. 2., völlig neu bearb. Aufl. 1981. Wv.

Band 18:
Wedekind, H./T. Härder
Datenbanksysteme II
430 S. mit Abb. 1976. Wv.

Band 19:
Kulisch, U.
Grundlagen des numerischen Rechnens. Mathematische Begründung der Rechnerarithmetik
467 S. 1976. Wv.

Band 20:
Zima, H.
Betriebssysteme. Parallele Prozesse
325 S. 2. Aufl. 1980. Wv.

Band 21:
Mies, P./D. Schütt
Feldrechner
150 S. 1976. Wv.

Band 22:
Denert, E./R. Franck
Datenstrukturen
362 S. 1977. Wv.

Band 23:
Ecker, K.
Organisation von parallelen Prozessen. Theorie deterministischer Schedules
280 S. 1977. Wv.

Band 24:
Kaucher, E./R. Klatte/Ch. Ullrich
Höhere Programmiersprachen ALGOL, FORTRAN, PASCAL in einheitlicher und übersichtlicher Darstellung
258 S. 1978. Wv.

Band 25:
Motsch, W.
Halbleiterspeicher.
Technik, Organisation und Anwendung
237 S. 1978. Wv.

Band 26:
Görke, W.
Mikrorechner.
Eine Einführung in ihre Technik und Funktion
251 S. 2., überarbeitete und erweiterte Aufl. 1980. Wv.

Band 27:
Mayer, O.
Syntaxanalyse
433 S. 2., durchgesehene und ergänzte Aufl. 1982. Wv.

Band 28:
Schrack, G.
Grafische Datenverarbeitung.
Eine Einführung
264 S. 1978. Wv.

Band 29:
Waller, H./P. Hilgers
Mikroprozessoren.
Vom Bauteil zur Anwendung
332 S. mit Abb. 1980. Wv.

Band 30:
Reusch, P.
Informationssysteme, Dokumentationssprachen, Data Dictionaries. Eine Einführung
216 S. 1980. Wv.

Band 31:
Ershov, A. P.
Einführung in die Theoretische Programmierung.
Gespräche über die Methode
Etwa 450 S. mit Abb. Aus dem Russischen. 1982. Wv.

Band 32:
Koch, G.
Maschinennahes Programmieren von Mikrocomputern
274 S. 1981. Wv.

Band 33:
Stetter, F.
Softwaretechnologie.
Eine Einführung
303 S. 1981. Wv.

Band 34:
Balzert, H.
Die Entwicklung von Software-Systemen.
Prinzipien, Methoden, Sprachen, Werkzeuge
Etwa 300 S. 1982. Wv.

Band 35:
Hotz, G./K. Estenfeld
Formale Sprachen.
Eine automatentheoretische Einführung
234 S. 1981. Wv.

Band 36:
Zima, H.
Compilerbau I.
Analyse
Etwa 350 S. 1982. Wv.

Physik

Baltes, H. P./E. R. Hilf
Spectra of Finite Systems
116 S. In englischer Sprache. 1976. Wv.

Barut, A. O.
Die Theorie der Streumatrix für die Wechselwirkungen fundamentaler Teilchen
Band I: Gruppentheoretische Beschreibung der S-Matrix.
225 S. mit Abb. Aus dem Englischen. 1971. (HTB 438)
Band II: Grundlegende Teilchenprozesse.
212 S. mit Abb. Aus dem Englischen. 1971. (HTB 555)

Bethge, K.
Quantenphysik.
Eine Einführung in die Atom- und Molekülphysik
271 S. 1978. Wv. Unter Mitarbeit von Dr. G. Gruber, Universität Frankfurt.

Bjorken, J. D./S. D. Drell
Relativistische Quantenmechanik
312 S. mit Abb. 1966. Aus dem Englischen. (HTB 98)

Bjorken, J. D./S. D. Drell
Relativistische Quantenfeldtheorie
409 S. Unveränderter Neudruck 1978. Aus dem Englischen. (HTB 101)

Bleuler, K./H. R. Petry/ D. Schütte (Hrsg.)
Mesonic Effects in Nuclear Structure
181 S. mit Abb. In englicher Sprache. 1975. Wv.

Blum, P./K. Schuchardt/V. Gärtner
Die Hochatmosphäre. Eine Einführung
Etwa 250 S. mit zahlr. Abb. 1982. Wv.

Bodenstedt, E.
Experimente der Kernphysik und ihre Deutung
Band I: 290 S. mit Abb. 2., durchgesehene Aufl. 1979. Wv.
Band II: XIV, 293 S. mit Abb. 2., durchgesehene Aufl. 1978. Wv.
Band III: 303 S. mit Abb. 2., durchgesehene Aufl. 1979. Wv.

Borucki, H.
Einführung in die Akustik
236 S. 2., durchgesehene Aufl. 1980. Wv.

Donner, W.
Einführung in die Theorie der Kernspektren
Band II: Erweiterung des Schalenmodells, Riesenresonanzen.
107 S. mit Abb. 1971. (HTB 556)

Eder, G.
Quantenmechanik I
273 S. 2. Aufl. 1980. Wv.

Eder, G.
Atomphysik. Quantenmechanik II
259 S. 1978. Wv.

Eder, G.
Elektrodynamik
273 S. 1967. (HTB 233)

Emendörfer, D./K. H. Höcker
Theorie der Kernreaktoren
Band 1: Der stationäre Reaktor.
Etwa 380 S. mit Abb. 2., neu bearbeitete Aufl. 1982. Wv.

Gasiorowicz, S.
Elementarteilchenphysik
742 S. mit 119 Abb. 1975. Aus dem Englischen. Wv.

Grott, S. R. de
Thermodynamik irreversibler Prozesse
216 S. mit 4 Abb. 1960. Aus dem Englischen. (HTB 18)

Groot, S. R. de/P. Mazur
Anwendung der Thermodynamik irreversibler Prozesse
349 S. 1974. Aus dem Englischen. Wv.

Haken, H.
Licht und Materie I. Elemente der Quantenoptik
155 S. 1979. Wv.

Haken, H.
Licht und Materie II. Laser
225 S. 1981. Wv.

Heisenberg, W.
Physikalische Prinzipien der Quantentheorie
117 S. mit 22 Abb. 1958. (HTB 1)

Henley, E. M./W. Thirring
Elementare Quantenfeldtheorie
336 S. mit Abb. 1975. Aus dem Englischen. Wv.

Hund, F.
Geschichte der physikalischen Begriffe
Teil I: Die Entstehung des mechanischen Naturbildes.
221 S. 2., neu bearbeitete Aufl. 1978. (HTB 543)
Teil II: Die Wege zum heutigen Naturbild.
233 S. 2., neu bearbeitete Aufl. 1978. (HTB 544)

Hund, F.
Geschichte der Quantentheorie
262 S. mit Abb. 2. Aufl. 1975. Wv.

Hund, F.
Grundbegriffe der Physik
Teil I: Makroskopische Vorgänge.
150 S. mit Abb. 2., neu bearbeitete Aufl. 1979. (HTB 449)
Teil II: Mikroskopischer Hintergrund.
151 S. mit Abb. 2., neu bearbeitete Aufl. 1979. (HTB 450)

Källen, G./J. Steinberger
Elementarteilchenphysik
687 S. mit Abb. 2., verbesserte Aufl. 1974. Aus dem Englischen. Wv.

Kippenhahn, R./C. Möllenhoff
Elementare Plasmaphysik
297 S. mit Abb. 1975. Wv.

Luchner, K.
Aufgaben und Lösungen zur Experimentalphysik
Band II: Elektromagnetische Vorgänge.
150 S. mit Abb. 1966. (HTB 156)
Band III: Grundlagen zur Atomphysik.
125 S. mit Abb. 1973. (HTB 157)

Lüscher, E.
Experimentalphysik
Band I: Mechanik, geometrische Optik, Wärme.
Band I/1. Teil: 260 S. mit Abb. 1967. (HTB 111)
Band I/2. Teil: 215 S. mit Abb. 1967. (HTB 114)
Band II: Elektromagnetische Vorgänge.
371 S. 2., überarb. Aufl. 1981. (HTB 115)

Lüst, R.
Hydrodynamik
234 S. 1978. Wv.

Mitter, H.
Elektrodynamik
391 S. 1980. (HTB 707)

Mitter, H.
Quantentheorie
313 S. mit Abb. 2. Aufl. 1979. (HTB 701)

Møller, C.
Relativitätstheorie
316 S. 1977. Wv.

Neff, H.
Physikalische Meßtechnik
160 S. mit Abb. 1976. (HTB 66)

Neuert, H.
Experimantalphysik für Mediziner, Zahnmediziner, Pharmazeuten und Biologen
292 S. mit Abb. 1969. (HTB 712)

Neuert, H.
Physik für Naturwissenschaftler
Band I: Mechanik und Wärmelehre.
173 S. 1977. (HTB 727)
Band II: Elektrizität und Magnetismus, Optik. 198 S. 1977. (HTB 728)
Band III: Atomphysik, Kernphysik, chemische Analyseverfahren.
326 S. 1978. (HTB 729)

Nitsch, J./J. Pfarr/ E.-W. Stachow (Hrsg.)
Grundlagenprobleme der modernen Physik
319 S. 1981. Wv.

Rollnik, H.
Teilchenphysik
Band I: Grundlegende Eigenschaften von Elementarteilchen.
Etwa 188 S. 2. Aufl. 1982. (HTB 706)
Band II: Innere Symmetrien der Elementarteilchen.
158 S. mit Abb. z. T. farbig. 1971. (HTB 759)

Rose, M. E.
Relativistische Elektronentheorie
Band I: 193 S. mit Abb. 1971. Aus dem Englischen. (HTB 422)
Band II: 171 S. mit Abb. 1971. Aus dem Englischen. (HTB 554)

Scherrer, P./P. Stoll
Physikalische Übungsaufgaben
Band I: Mechanik und Akustik.
96 S. mit 44 Abb. 1962. (HTB 32)
Band II: Optik, Thermodynamik, Elektrostatik.
103 S. mit Abb. 1963. (HTB 33)
Band III: Elektrizitätslehre, Atomphysik.
103 S. mit Abb. 1964. (HTB 34)

Schütte, D./K. Holinde/ K. Bleuler (Hrsg.)
The Meson Theory of Nuclear Forces and Nuclear Matter
393 S. 1980. Wv.

Seiler, H.
Abbildungen von Oberflächen mit Elektronen, Ionen und Röntgenstrahlen
131 S. mit Abb. 1968. (HTB 428)

Sexl, R. U./H. K. Urbantke
Gravitation und Kosmologie. Eine Einführung in die Allgemeine Relativitätstheorie
Etwa 370 S. 2., überarb. und erweiterte Aufl. 1982. Wv.

Teichmann, H.
Einführung in die Atomphysik
135 S. mit 47 Abb. 3. Auflage 1966. (HTB 12)

Teichmann, H.
Halbleiter
156 S. mit 55 Abb. 3. Auflage 1969. (HTB 21)

Wagner, C.
Methoden der naturwissenschaftlichen und technischen Forschung
219 S. mit Abb. 1974. Wv.

Weizel, W.
Einführung in die Physik
Band III: Optik und Atomphysik.
194 S. mit 99 Abb. 5. Auflage 1963. (HTB 5)

Weizel, W.
Physikalische Formelsammlung
Band II: Optik, Thermodynamik, Relativitätstheorie.
148 S. 1964. (HTB 36)
Band III: Quantentheorie.
196 S. mit zweifarbigem Druck der Formeln. 1966. (HTB 37)

Zimmermann, P.
Eine Einführung in die Theorie der Atomspektren
91 S. mit Abb. 1976. Wv.

Astronomie

Becker, F.
Geschichte der Astronomie
201 S. 4. Aufl. 1980. Wv.

Scheffler, H./H. Elsässer
Bau und Physik der Galaxis
Etwa 650 S. 1982. Wv.

Scheffler, H./H. Elsässer
Physik der Sterne und der Sonne
535 S. mit Abb. 1974. Wv.

Schneider, M.
Himmelsmechanik
480 S. mit Abb. 2., verbesserte Aufl. 1981. Wv.

Schurig, R./P. Götz/K. Schaifers
Himmelsatlas (Tabuale caelestes)
44 S. 8. Aufl. 1960. Wv.

Voigt, H. H.
Abriß der Astronomie
558 S. mit Abb. 3., überarbeitete Aufl. 1980. Wv.

Philosophie

Enzyklopädie Philosophie und Wissenschaftstheorie in 3 Bänden
Herausgegeben von Jürgen Mittelstraß. Rund 4000 Stichwörter auf etwa 2400 Seiten. Wv.

Bunge, M.
Epistemologie. Aktuelle Fragen der Wissenschaftstheorie
Etwa 240 S. 1982. Aus dem Spanischen. Wv.

Haas, G.
Konstruktive Einführung in die formale Logik
Etwa 200 S. 1982. Wv.

Kamlah, W.
Philosophische Anthropologie. Sprachkritische Grundlegung und Ethik
192 S. 1973. (HTB 238)

Kamlah, W.
Von der Sprache zur Vernunft. Philosophie und Wissenschaft in der neuzeitlichen Profanität
230 S. 1975. Wv.

Kamlah, W./P. Lorenzen
Logische Propädeutik. Vorschule des vernünftigen Redens
239 S. 2., verb. und erweiterte Aufl. 1973. (HTB 227)

Kanitscheider, B.
Vom absoluten Raum zur dynamischen Geometrie
139 S. 1976. Wv.

Leinfellner, W.
Einführung in die Erkenntnis- und Wissenschaftstheorie
227 S. 3. Aufl. 1980. (HTB 41)

Lorenzen, P.
Normative Logic and Ethics
89 S. 1969. In englischer Sprache. (HTB 236)

Lorenzen, P./O. Schwemmer
Konstruktive Kogik, Ethik und Wissenschaftstheorie
331 S. mit Abb. 2., verbesserte Aufl. 1975. (HTB 700)

Mittelstaedt, P.
Philosophische Probleme der modernen Physik
227 S. 6., durchgesehene Aufl. 1981. (HTB 50)

Mittelstaedt, P.
Die Sprache der Physik
139 S. 1972. Wv.

Reihe: Grundlagen der exakten Naturwissenschaften

Herausgegeben von Prof. Dr. Peter Mittelstaedt, Universität Köln.

Band 1:
Mittelstaedt, P./J. Pfarr (Hrsg.)
Grundlagen der Quantentheorie
159 S. 1980. Wv.

Band 2:
Strohmeyer, I.
Transzententalphilosophische und physikalische Raum-Zeit-Lehre
184 S. 1980. Wv.

Band 3:
Mittelstaedt, P.
Der Zeitbegriff in der Physik
188 S. 2., verbesserte und erweiterte Aufl. 1980. Wv.

Band 4:
Pfarr, J. (Hrsg.)
Protophysik und Relativitätstheorie
240 S. 1981. Wv.

Band 5:
Neumann, H. (Hrsg.)
Interpretations and Foundations of Quantum Theory
144 S. 1981. In englischer Sprache. Wv.

Chemie

Grimmer, G.
Biochemie
376 S. mit Abb. 1969. (HTB 187)

Kaiser, R.
Chromatographie in der Gasphase
Band I: Gas-Chromatographie.
220 S. mit 81 Abb. 2. Aufl. 1973. (HTB 22)
Band IV/2. Teil: Quantitative Auswertung.
118 S. mit Abb. 2., erweiterte Aufl. 1969. (HTB 472)

Laidler, K. J.
Reaktionskinetik
Band I: Homogene Gasreaktionen.
216 S. mit Abb. 1970. Aus dem Englischen.
(HTB 290)

Preuß, H.
Quantentheoretische Chemie
Band I: Die halbempirischen Regeln.
94 S. mit 19 Abb. 1963. (HTB 43)
Band II: Der Übergang zur Wellenmechanik, die allgemeinen Rechenverfahren.
238 S. mit 19 Abb. 1965. (HTB 44)
Band III: Wellenmechanische und methodische Ausgangspunkte.
222 S. mit Abb. 1967. (HTB 45)

Riedel, L.
Physikalische Chemie.
Eine Einführung für Ingenieure
406 S. 1974. Wv.

Schmidt, M.
Anorganische Chemie
Band I: Hauptgruppenelemente.
301 S. mit Abb. 1967. (HTB 86)
Band II: Übergangsmetalle. 221 S. mit Abb. 1969. (HTB 150)

Medizin

Forth, W./D. Henschler/
W. Rummel (Hrsg.)
Allgemeine und spezielle Pharmakologie und Toxikologie
Für Studenten der Medizin, Veterinärmedizin, Pharmazie, Chemie, Biologie sowie für Ärzte und Apotheker.
3., überarbeitete Aufl. 1980. 688 S. Über 400 meist zweifarbige Abb. sowie mehr als 280 Tabellen. Wv.

Haas, H.
Ursprung, Geschichte und Idee der Arzneimittelkunde
(Reihe: Pharmakologie und Toxikologie, Band 1)
178 S. 1981. Wv.

Ingenieurwissenschaften

Billet, R.
Grundlagen der thermischen Flüssigkeitszerlegung
150 S. mit 50 Abb. 1962. (HTB 29)

Billet, R.
Optimierung in der Rektifiziertechnik unter besonderer Berücksichtigung der Vakuumrektifikation
129 S. mit Abb. 1967. (HTB 261)

Billet, R.
Trennkolonnen für die Verfahrenstechnik
151 S. mit Abb. 1971. (HTB 548)

Böhm, H.
Einführung in die Metallkunde
236 S. mit Abb. 1968. (HTB 196)

Bosse, G.
Grundlagen der Elektrotechnik
Band I: Das elektrostatische Feld und der Gleichstrom. Unter Mitarbeit von W. Mecklenbräuker. 141 S. mit Abb. 1966. (HTB 182)
Band II: Das megnetische Feld und die elektromagnetische Induktion. Unter Mitarbeit von G. Wiesemann.
154 S. mit Abb. 2., überarbeitete Aufl. 1978. (HTB 183)
Band III: Wechselstromlehre, Vierpol- und Leitungstheorie. Unter Mitarbeit von A. Glaab.
135 S. 2., überarbeitete Aufl. 1978. (HTB 184)
Band IV: Drehstrom, Ausgleichsvorgänge in linearen Netzen. Unter Mitarbeit von J. Hagenauer.
164 S. mit Abb. 1973. (HTB 185)

Eschenauer, H./W. Schnell
Elastizitätstheorie I.
Grundlagen, Scheiben und Platten
256 S. 1981. Wv.

Feldtkeller, E.
Dielektrische und magnetische Materialeigenschaften
Band I: Meßgrößen, Materialübersicht und statistische Eigenschaften.
242 S. mit Abb. 1973. (HTB 485)

Glaab, A./J. Hagenauer
Übungen in Grundlagen der Elektrotechnik III, IV
228 S. mit Abb. 1973. (HTB 780)

Gross, D./W. Schnell
Formel- und Aufgabensammlung zur Technischen Mechanik II. Elastostatik
180 S. mit Abb. 1980. (HTB 792)

Klein, W.
Vierpoltheorie
159 S. mit Abb. 1972. Wv.

Mahrenholtz, O.
Analogrechnen in Maschinenbau und Mechanik
208 S. mit Abb. 1968. (HTB 154)

Marguerre, K./H. Wölfel
Technische Schwingungslehre. Lineare Schwingungen vielgliedriger Strukturen
338 S. 1979. Wv.

Marguerre, K./H.-T. Woernle
Elastische Platten
242 S. mit 125 Abb. 1975. Wv.

Mesch, F.
Meßtechnisches Praktikum
Für Maschinenbauer und Verfahrentechniker.
217 S. mit Abb. 3., überarbeitete Aufl. 1981. (HTB 736)

Pestel, E.
Technische Mechanik
Band I: Statik.
Etwa 280 S. 2., neu bearb. und ergänzte Aufl. 1982. Wv.

Pestel, E./J. Wittenburg
Technische Mechanik
Band 2: Festigkeitslehre.
441 S. mit über 350 Abb. 1981. Wv.

Piefke, G.
Feldtheorie
Band I: Maxwellsche Gleichungen, Elektrostatik, Wellengleichung, verlustlose Leitungen.
264 S. Verbesserter Nachdruck 1977. (HTB 771)
Band II: Verlustbehaftete Leitungen, Grundlagen der Antennenabstrahlung, Einschwingvorgang.
231 S. mit Abb. 1973. (HTB 773)
Band III: Beugungs- und Streuprobleme, Wellenausbreitung in anisotropen Medien.
362 S. 1977. (HTB 782)

Sagirow, P.
Satellitendynamik
191 S. 1970. (HTB 719)

Schnell, W./D. Gross
Formel- und Aufgabensammlung zur Technischen Mechanik I. Statik
180 S. mit Abb. 1979. (HTB 791)

Schnell, W./D. Gross
Formel- und Aufgabensammlung zur Technischen Mechanik III. Kinetik
180 S. mit Abb. 1980. (HTB 793)

Stüwe, H. P.
Einführung in die Werkstoffkunde
197 S. mit Abb. 2., verbesserte Aufl. 1978. (HTB 467)

Stüwe, H. P./G. Vibrans
Feinstrukturuntersuchungen in der Werkstoffkunde
138 S. mit Abb. 1974. Wv.

Troost, A.
Einführung in die allgemeine Werkstoffkunde metallischer Werkstoffe I
507 S. mit Abb. 1980. Wv.

Waller, H./W. Krings
Matrizenmethoden in der Maschinen- und Bauwerksdynamik
377 S. mit 159 Abb. 1975. Wv.

Wasserrab, Th.
Gaselektronik
Band I: Atomtheorie.
223 S. mit Abb. 1971. (HTB 742)
Band II: Niederdruckentladungen, Technik der Gasentladungsventile.
230 S. mit Abb. 1972. (HTB 769)

Wiesemann, G.
Übungen in Grundlagen der Elektrotechnik II
202 S. mit Abb. 1976. (HTB 779)

Wiesemann, G./W. Mecklenbräuker
Übungen in Grundlagen der Elektrotechnik I
179 S. mit Abb. 1973. (HTB 778)

Wolff, I.
Grundlagen und Anwendungen der Maxwellschen Theorie
Band I: Mathematische Grundlagen, die Maxwellschen Gleichungen, Elektrostatik.
236 S. mit Abb. 1968. (HTB 818)
Band II: Strömungsfelder, Magnetfelder, quasistationäre Felder, Wellen.
263 S. mit Abb. 1970. (HTB 731)

Reihe: Theoretische und experimentelle Methoden der Regelungstechnik

Herausgegeben von Gerhard Preßler, Hartmann & Braun, Frankfurt.

Band 1:
Preßler, G.
Regelungstechnik
348 S. mit 235 Abb. 3., überarbeitete Aufl. 1967. (HTB 63)

Band 4:
Klefenz, G.
Die Regelung von Dampfkraftwerken
Etwa 250 S. 3., überarbeitete Aufl. 1982. Wv.

Band 8/9:
Starkermann, R.
Die harmonische Linearisierung
Band II: 83 S. mit Abb. 1970. (HTB 470)

Band 10:
Starkermann, R.
Mehrgrößen-Regelsysteme
Band I: 173 S. mit Abb. 1974. Wv.

Band 12:
Schwarz, H.
Optimale Regelung linearer Systeme
242 S. mit Abb. 1976. Wv.

Band 13:
Latzel, W.
Regelung mit dem Prozeßrechner (DDC)
213 S. mit über 100 Abb. 1977. Wv.

Reihe: Gesellschaft, Recht, Wirtschaft

Herausgegeben von Prof. Dr. Eduard Gaugler, Dr. Wolfgang Goedecke, Prof. Dr. Heinz König, Prof. Dr. Günther Wiese, Prof. Dr. Rudolf Wildenmann, Universität Mannheim.

Band 1:
Albert, H./M. C. Kemp/ W. Krelle/G. Menges/ W. Meyer
Ökonometrische Modelle und sozialwissenschaftliche Erkenntnisprogramme
111 S. 1978. Wv.

Band 2:
Bogaert, R./P. C. Hartmann
Essays zur historischen Entwicklung des Bankensystems
48 S. 1980. Wv.

Band 3:
Nerlove, M./S. Heiler/ H.-J. Lenz/B. Schips/ H. Garbers
Problems of Time Series Analysis
140 S. 1980. Wv.

Band 4:
Aumann, R. J./J. C. Harsanyi/ W. Hildenbrand/M. Maschler/ M. A. Perles/J. Rosenmüller/ R. Selten/M. S. Shubik/ G. L. Thompson
Essays in Game Theory and Mathematical Economics in Honor of Oskar Morgenstern
200 S. 1981. In englischer Sprache. Wv.

Band 5:
Steinmann, H./G. Gäfgen/ W. Blomeyer
Die Kosten der Mitbestimmung
139 S. 1981. Wv.

Band 6:
Frisch, W./E. v. Caemmerer/ G. Wüst/F. A. Mann/R. Fischer
In Memoriam Konrad Duden
Theorie und Praxis im Wirtschaftsrecht
75 S. 1982. Wv.

Band 7:
Braun, W./K.-F. Ackermann/ E. Gaugler/H. Kossbiel/ G. Reber/W. Weber
Verantwortliche Personalführung
128 S. 1982. Wv.

Geographie/Geologie

Ganssen, R.
Grundsätze der Bodenbildung
135 S. mit 19 Zeichnungen und einer mehrfarbigen Tafel. 1965. (HTB 327)

Gierloff-Emden, H.-G./ H. Schroeder-Lanz
Luftbildauswertung
Band I: Grundlagen.
154 S. mit Abb. 1970. (HTB 358)

Kertz, W.
Einführung in die Geophysik
Band I: Erdkörper.
232 S. mit Abb. 1969. (HTB 275)
Band II: Obere Atmosphäre und Magnetosphäre.
210 S. mit Abb. 1971. (HTB 535)

Möller, F.
Einführung in die Meteorologie
Band I: Meteorologische Elementarphänomene.
222 S. mit Abb. und 6 Farbtafeln. 1973. (HTB 276)
Band II: Komplexe meteorologische Phänomene.
223 S. mit Abb. 1973. (HTB 288)

Wunderlich, H.-G.
Bau der Erde.
Geologie der Kontinente und Meere
Band II: Asien, Australien.
164 S., 16 S. farbige Abb. 1975. Wv.

Wunderlich, H.-G.
Einführung in die Geologie
Band I: Exogene Dynamik.
214 S. mit ca. 50 Abb. und 24 farbigen Bildern. 1968. (HTB 340)
Band II: Endogene Dynamik.
231 S. mit Abb. und 16 farbigen Bildern. 1968. (HTB 341)

B.I.-Hochschulatlanten

Dietrich, G./J. Ulrich (Hrsg.)
Atlas zur Ozeanographie
76 S. 1968. (HTB 307)

Schmithüsen, J. (Hrsg.)
Atlas zur Biogeographie
80 S. 1976. (HTB 303)

Wagner, K. (Hrsg.)
Atlas zur physischen Geographie (Orographie)
59 S. 1971. (HTB 304)

Edgar Kaucher
Geboren am 8. Januar 1945. 1971 Diplom, 1973 Promotion, 1981 Habilitation; seit 1972 wissenschaftlicher Mitarbeiter am Institut für Angewandte Mathematik der Universität Karlsruhe.

Rudi Klatte
Geboren am 19. Januar 1946. 1970 Diplom, 1975 Promotion; seit 1971 wissenschaftlicher Mitarbeiter am Institut für Angewandte Mathematik der Universität Karlsruhe.

Christian Ullrich
Geboren am 16. Oktober 1944. 1972 Promotion, 1977 Habilitation; seit 1980 Professor für Angewandte Mathematik an der Universität Karlsruhe.

Jürgen Wolff v. Gudenberg
Geboren am 16. Juni 1952. 1976 Diplom, 1980 Promotion; seit 1982 Hochschulassistent am Institut für Angewandte Mathematik der Universität Karlsruhe.

Dieses Buch richtet sich an alle, die die Programmiersprache ADA erlernen und anwenden wollen. Durch die einprägsame graphische Darstellung der Syntax zusammen mit der präzisen Beschreibung der Semantik und der didaktischen Aufbereitung des Stoffes vom einfachen Sprachelement bis hin zum komplizierten Programm ist es gleichermaßen geeignet zum Selbststudium, als Begleittext zu Vorlesungen und als Nachschlagewerk. Wegen der hohen Komplexität der Programmiersprache ADA wird das Verständnis durch die Kenntnis einer anderen Programmiersprache ganz wesentlich erleichtert. In derselben Reihe erscheinen Bände über BASIC, FORTRAN und PASCAL. Weitere Bände sind in Vorbereitung.

ISBN 3-411-00607-2